TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TESTIVETOISEN OHJELMISTOKEHITYKSEN PILOTOINTI VAKUUTUSALAN OHJELMISTOTUOTANNOSSA

Diplomityö

Tarkastaja: professori Tarmo Lipping
Tarkastaja ja aihe hyväksytty
25. kesäkuuta 2018

TIIVISTELMÄ

MARI MEUSEL: Testivetoisen ohjelmistokehityksen pilotointi vakuutusalan ohjelmistotuotannossa
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua, 0 liitesivua
Elokuu 2018
Johtamisen ja tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto ja tiedonhallinta
Tarkastaja: professori Tarmo Lipping

Avainsanat: TDD, testivetoinen ohjelmistokehitys, edut, kehityssykli

Testivetoinen ohjelmistokehitys (Test-Driven Development, TDD) on kehitystapa, jossa yksikkötestit kirjoitetaan ennen uuden toiminnallisuuden ohjelmointia. TDD-kehitystavalle on esitetty useita etuja verrattuna tapaan, jossa yksikkötestit kirjoitetaan vasta ohjelmointityön jälkeen. Tässä päättötyössä TDD-kehitystapaa pilotoitiin vakuutusalan ohjelmistotuotannossa ja arvioitiin kehitystavan etuja ja soveltuvuutta päättötyön tilanneen yrityksen ohjelmistokehitysympäristöön ja toimintatapoihin. Kehitystapaa käytettiin tilaajan kahden oman ohjelmistotuotteen välisen rajapinnan toteutuksessa.

TDD-kehitystavan esitettyjä etuja on hankala verrata muihin ohjelmistokehitystapoihin, koska vertailukelpoisten tulosten saamiseksi sama toiminnallisuus pitäisi ohjelmoida kahdella eri kehitystavalla. Jos yksikkötestit toteutetaan yhtä kattavasti millä tahansa kehitysmenetelmällä, useat TDD-menetelmän eduiksi esitetyt väitteet toteutuvat käytetystä menetelmästä riippumatta. Pilotoinnin perusteella menetelmän eduiksi voidaan todentaa, että TDD-kehitystavan käyttö ohjaa syntyvää ohjelmistokoodia yksikkötestikelpoiseksi sekä nopeuttaa ohjelmistotyöskentelyä, kun työn alla olevaa kooditoteutusta pääsee nopeasti testaamaan yksikkötestien avulla.

Pilotoinnin sekä päättötyön tilaajan prosessikehitystiimin työskentelyn tuloksena päädyttiin lisäämään yksikkötestien määrää ohjelmistotuotteiden olemassa oleviin, liiketoiminnan kannalta kriittisimpiin osuuksiin. Tilaajan sisäiset vaatimukset uusien toiminnallisuuksien yksikkötestien kattavuudesta selkeytetään sekä niiden toteutumista valvotaan koodikatselmointien yhteydessä. Kehityskeskusteluissa kävi myös ilmi, että sisäistä koulutusta yksikkötestien toteuttamiselle tarvitaan ja sellaista myös aiotaan järjestää nykyisille ja uusille ohjelmistokehittäjille. Ohjelmistokehittäjän omaan harkintaan jää, mitä ohjelmistokehitystapaa haluaa käyttää.

ABSTRACT

MARI MEUSEL: A pilot study of Test-Driven software Development in insurance industries software production
Tampere University of Technology
Master of Science Thesis, 45 pages, 0 Appendix pages
August 2018
Master's Degree Programme in Management and Information Technology
Major: Software Engineering and Data Management
Examiner: Professor Tarmo Lipping

Keywords: TDD, Test-Driven software Development, benefits, development cycle

Test-Driven Development (TDD) is a software development method whereby unit tests are written before the actual functionalities or features of the software. TDD has been proclaimed to carry many benefits, as opposed to the software development method where unit tests are written in the post-coding phase. This thesis pilots the TDD in a software development company whose main clients operate in the insurance business. The thesis evaluates the benefits of TDD and its suitability to the software development environment and practices of the pilot organization. TDD was used in creating an interface between two software products of the pilot organization.

The alleged benefits of TDD are hard to compare against other software development methods. In order to get fully transparent and comparable results the same functionality should be written multiple times; once with every method. In cases where unit tests are executed with a high level of rigour and detail, most of the benefits awarded to TDD will also be true with other development methods. What the pilot showed was that TDD as a development method organically shapes the code towards a more unit test-friendly form as well as speeding up the overall development process by enabling quick tests of a written code already at the draft stage.

As a result of the pilot and the work done by the process development team of the pilot organization, the decision was made to increase the number of unit tests of the current software products' most critical parts. What is more, the pilot organization's internal guidelines for unit testing new functionalities are to be clarified and then followed through in code review meetings. Development discussions with the personnel also showed that internal training for executing unit tests is required and such training is to be organized for current and new software developers. However, in the end it is up to the software developer to decide which development method to use.

ALKUSANAT

Aloitin ohjelmistoinsinöörin työt Profit Software Oy:ssä, jossa työn yhtenä tarkoituksena oli saattaa opintoni Tampereen teknillisessä yliopistossa loppuun päättötyön muodossa. Aloittaessani lähes kaikki yhtiössä käytetyt tekniikat olivat minulle uusia, joten muutama ensimmäinen kuukausi kului tekniikoihin ja yrityksen ohjelmistoihin perehtymisessä. Haluan kiittää tehokkaasta työhön perehdytyksestä erityisesti Profit Software Oy:n minulle osoittamaa tutoria Jarmo Hiljasta kärsivällisyydestä ja hyvästä ohjauksesta. Kun päättötyöhön liittyvän ohjelmistotyön toteutus alkoi, olin ehtinyt saada riittävät valmiudet työn toteuttamiselle.

Haluan kiittää Profit Software Oy:tä mahdollisuudesta päättötyön tekoon sekä yrityksen kaikkia päättötyöhön liittyneitä henkilöitä, erityisesti työn tarkastusvaiheessa avuksi olleita Jonna Lainetta ja Teppo Eevaa. Ystävät ja sukulaiset ovat olleet henkisenä tukena, silloin kun työ ei ole tuntunut edistyvän ja usko on tuntunut loppuvan. Suuri kiitos myös heille.

Porissa 12.8.2018

Mari Meusel

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. TESTAUSMENETELMÄT OHJELMISTOKEHITYSTYÖSSÄ 4
2.1 Yksikkötestaus 6
2.2 Yksikkötestien asettaminen ja annotaatiot 7
2.3 Mock-objektit 10
2.4 Ohjelmiston lähdekoodin refaktorointi 13
2.5 Testivetoinen ohjelmistokehitys 15
2.5.1 Testivetoinen ohjelmistokehityssykli 15
2.5.2 TDD-kehitysmenetelmän edut 17
3. PROFIT PLP JA COM -OHJELMISTOT JA OHJELMOINTIRAJAPINNAT 19
3.1 Profit Life & Pension, PLP 20
3.2 Claim Payout Manager, COM 21
3.3 COM ja PLP -ohjelmistojen kehitystarpeet 21
3.4 COM-ohjelmistossa käytetyt rajapintatekniikat 22
3.4.1 XML-rajapinta 22
3.4.2 JAXB-ohjelmointirajapinta 24
4. RAJAPINNAN TOTEUTUS, TESTAUS JA TESTITULOKSET 27
4.1 Ohjelmistokehitys- ja yksikkötestiajoympäristö 28
4.1.1 Testien suorittaminen ohjelmistokehityksen aikana 28
4.1.2 Testien suorittaminen ohjelmiston kääntämisen yhteydessä 31
4.2 TDD-kehitysmenetelmän toteuttaminen käytännön ohjelmointityössä 33
4.2.1 Yksikkötestien toteutus 33
4.2.2 Työskentely TDD-menetelmällä 35
4.3 Johtopäätökset 39
5. YHTEENVETO 42
LÄHTEET 44

LYHENTEET JA MERKINNÄT

API	Application Programming Interface, ohjelmointirajapinta
COM	Claim Payout Manager
IDE	Integrated Development Environment
JAXB	Java Architecture for XML Binding
PLP	Profit Life & Pension
TLD	Test-Last Development
TDD	Test-Driven Development, testivetoinen ohjelmistokehitys
UTF-8	compressed Unicode
XML	eXtensible Markup Language

1. JOHDANTO

Ohjelmiston laadulla tarkoitetaan ohjelmiston käytettävyyttä ja ohjelmiston toimimista oikein. Ohjelmiston laatua mitataan myös ohjelmiston ylläpidettävyystekijöillä, jolloin laatunäkökulmaa katsotaan ohjelmistotuotannon näkökulmasta. Kun ohjelmisto on käytössä vuosia ja sen elinkaari on pitkä, ohjelmistoa on tarpeen kehittää ja ylläpitää. Silloin kehityskustannukset saattavat nousta suuriksi, ellei ohjelmiston sisäisestä eli ohjelmakoodin laadusta ole jatkuvasti huolehdittu kehityksen eri vaiheissa. Ohjelmiston toiminnallisuuksien kasvaessa ohjelmistokehittäjiä tarvitaan enemmän ja kehittäjät saattavat vaihtua projektien aikana. Tekijöiden vaihtuessa ja ohjelmiston toiminnallisuuksien lisääntyessä ja monipuolistuessa koodin selkeys ja luettavuus nousevat avainasemaan. Uusia toimintoja suunniteltaessa ja toteutettaessa ohjelmistokehittäjän pitää ymmärtää, mitä ohjelmiston jokin toiminto tekee nykyisessä muodossaan, tehdä muutos tai uusi toiminnallisuus ja varmistua siitä, että muutos ei aiheuttanut ongelmia olemassa oleviin toimintoihin tai rakenteisiin.

Tämän päättötyön keskiössä olevat vakuutusalan ohjelmistot ovat asiakasyrityksille pitkäaikainen hankinta ja kyseiset ohjelmistot ovat asiakkaan käytössä useita vuosia eteenpäin. Tämän päättötyön tilaajana toimivan yrityksen omia ohjelmistotuotteita on kehitetty useita vuosia ja niitä kehitetään jatkuvasti lisää omana tuotekehityksenä sekä asiakastarvelähtöisesti. Kun asiakas on ottanut ohjelmiston käyttöön ja ohjelmistoon tulee tarve saada asiakaskohtaisesti räätälöityjä lisätoiminnallisuuksia tai koodikorjauksia, tehdään ohjelmistopäivitys päivityspaketin muodossa. Asiakas voi tehdä myös koko ohjelmiston version noston uudempaan, jolloin kyseessä on suurempi päivitysprosessi. Koska vakuutussopimusten hallintaan käytettävien ohjelmistotuotteiden elinkaari on pitkä, ohjelmiston toimittajan pitää pystyä takaamaan niiden ylläpidettävyys ja jatkokehitettävyys asiakkaan käyttövuosien aikana. Toimintojen virheettömyys on tärkeää etenkin, kun kyseisillä ohjelmistoilla käsitellään ihmisten rahoja vakuutus- ja eläkesopimusten sekä korvausprosessien hallinnassa. Ohjelmiston kehityskaaren aikana pitää varmistaa, ettei ohjelmiston kompleksisuus kasva hallitsemattomaksi kokonaisuudeksi. Jos näin käy, vaarana on, että ohjelmistokehittäjät eivät enää uskalla tehdä muutoksia monimutkaisten koodirakenteiden takia tai jatkokehityksen kustannukset nousevat liian korkeiksi.

Ohjelmiston laadun valvontaan on käytössä runsaasti erilaisia testausmenetelmiä ja eri menetelmillä on omat käyttötarkoituksensa. Tässä päättötyössä tarkastellaan laatunäkökulmaa ohjelmistokehittäjän katsontakulmasta ja sen vuoksi työssä tutustutaan syvällisemmin yksikkötestausmenetelmiin, joita tarkastellaan tarkemmin päättötyön luvussa 2 *Testausmenetelmät ohjelmistokehitys*. Yksikkötesteissä ohjelmistokehittäjä suunnittelee

ja toteuttaa testejä, joista kukin testaa työn alla olevan ohjelmaosion pientä toiminnallisuutta, yleensä ohjelman luokkaa tai metodia. Jokainen yksikkötesti pyritään toteuttamaan riippumattomaksi muista ohjelmiston toiminnallisuuksista, jolloin yksittäinen testi paljastaa vain kyseisen, testattavan toiminnallisuuden puutteet ja ongelmat. Yksikkötestit kertyvät ohjelmiston testauspuolelle ohjelmiston kehitystyön aikana, jolloin kaikki aiemmissa kehitysvaiheissa toteutetut testit säilyvät mukana ohjelmistokokonaisuudessa. Kun kaikki eri kehityskaaren vaiheissa toteutetut sekä vasta valmistuneet yksikkötestit suoritetaan uusien ominaisuuksien tuotannossa, huomataan, jos uudet toiminnallisuudet aiheuttavat ongelmia aiempiin toiminnallisuuksiin. Ohjelmistokehittäjä voi luottavaisin mielin tehdä muutoksia, mikäli yksikkötestit on toteutettu tunnollisesti ohjelmiston kehityskaaren eri vaiheissa ja yksikkötestien kattavuus on riittävä. Jos uusi ominaisuus rikkoo jonkin aiemmin toteutetun ominaisuuden, yksikkötestit paljastavat ongelman ennen kuin ohjelmakoodi siirtyy tuotannossa eteenpäin.

Perinteisesti ohjelmistokehittäjä kirjoittaa yksikkötestit joko koko toiminnallisuuden tai pienempien koodiosuuksien ohjelmoinnin jälkeen. Tätä ohjelmointitapaa kutsutaan TLD-menetelmäksi (Test-Last Development). Testivetoinen ohjelmistokehitys (Test-Driven Development, TDD) tarkoittaa ohjelmistokehityksessä tapaa, jossa yksikkötestit kirjoitetaan ennen ohjelman uuden toiminnallisuuden toteuttamista. TDD-menetelmää noudatettaessa ohjelman määrittelyiden mukaiset toiminnot pilkotaan pieniin osiin, ja jokaiseen osaan kirjoitetaan testitapaukset ennen varsinaista kooditoteutusta. Testivetoinen ohjelmistokehitys on enemmänkin ketterä ohjelmistokehitystapa kuin testausmenetelmä, koska ennakolta suunnitellut testit ohjaavat syntyvän koodin rakennetta ja kehittäjän työskentelytapaa.

TDD-menetelmää hyödyntävässä ohjelmistokehitystyössä noudatetaan kehitystavalle tyypillistä kehityssykliä. Työskentelyprosessissa valitaan ensin ohjelmistomäärittelystä pieni toiminto ja toteutetaan toimintoon liittyvät yksikkötestit. Koska varsinaista toimintoa ei vielä ole, varmistetaan, että testit eivät mene läpi suorittamalla testit. Seuraavassa vaiheessa toteutetaan ohjelman toiminto ja vältetään toteuttamasta muita toiminnallisuuksia yksikkötestien ulkopuolelta. Kun testit menevät läpi, siistitään ohjelmistokoodi ja parannetaan ohjelmiston sisäistä rakennetta muuttamatta kuitenkaan ohjelman toiminnallisuutta. Tämän jälkeen ajetaan testit uudelleen ja todennetaan, että ohjelma toimii edelleen oikein. Kierros aloitetaan alusta taas uuden toiminnon toteutuksen alussa.

Testivetoisen ohjelmistokehityksen eduiksi on alan kirjallisuudessa ja artikkeleissa (Acharya, 2013; Martin, 2007) mainittu, että TDD-menetelmän avulla tuotetaan puhdasta, testikelpoista ja ylläpidettävää ohjelmakoodia. Kehitysmenetelmä myös dokumentoi oman toiminnallisuutensa ohjelmistotestien muodossa. Testejä tarkastelemalla ohjelmistokehittäjän on helppo päätellä, miten ohjelman kyseistä toimintoa käytetään. Kun ohjelmistoa kehitetään inkrementaalisesti, kehitystyö on helpompaa, kun koodia kirjoitetaan ja käytetään samanaikaisesti. TDD-menetelmän etuja on tarkasteltu tarkemmin luvussa 2.5.2 *TDD-kehitysmenetelmän edut*.

Tämän päättötyön yhteydessä toteutetaan tilaajan kahden oman ohjelmistotuotteen välille löyhä rajapinta, jossa tietoja siirretään yksisuuntaisesti kahden eri järjestelmän välillä eräajopohjaisesti. Profit Life & Pension (PLP) -ohjelmistossa hallitaan vakuutustuotteiden, tässä tapauksessa säästö- ja eläkevakuutusten, säästön aikaiset toiminnot. PLP-ohjelmistolla hallitaan myös vakuutuksen korvaustoiminnot, joilla tarkoitetaan sopimukselle kertyneiden rahojen ulosmaksua asiakkaalle. Korvaustoiminnoille halutaan jatkossa tarjota PLP-ohjelmiston omien toimintojen rinnalle vaihtoehto, jossa korvausten ulosmaksatus tapahtuukin toisen ohjelmiston kautta. Tilaajan toinen tähän päättötyöhön liittyvä oma ohjelmistotuote Claim Payout Manager (COM) on kehitetty korvausmaksatuksen toimintoihin ja ohjelmisto sisältää nykyisellään rajapintamäärittelyt muista ohjelmista peräisin olevien ulosmaksutietojen sisäänlukuun. Päättötyön luvussa 3 *Profit PLP ja COM -ohjelmistot ja ohjelmointirajapinnat* kerrotaan tarkemmin toteutukseen liittyvistä ohjelmistoista sekä kuvataan korvausmaksatusten siirtotekniikka tarkemmin.

Päättötyön ohjelmointiosuudessa toteutetaan PLP-ohjelmistoon COM:in rajapinnan toteuttavat toiminnallisuudet korvausten ulosmaksatuksen osalta käyttäen TDD-kehitysmenetelmää. Toteutustyöhön liittyvät raportit sekä oivallukset esitellään luvussa 4 *Rajapinnan toteutus, testaus ja testitulokset*. TDD-kehitystapaa ei aiemmin ole suunnitelmallisesti arvioitu tai suositeltu käytettäväksi ohjelmointityöskentelyssä yrityksen taholta. Jotkut päättötyön tilanneen yrityksen palveluksessa olleet tai olevat ohjelmistokehittäjät työskentelevät tai ovat työskennelleet TDD-menetelmällä, mutta kyseessä on ollut kyseisten kehittäjien oma suuntautuminen ja kiinnostuneisuus, ei niinkään yrityksen linjaus.

2. TESTAUSMENETELMÄT OHJELMISTOKEHITYSTYÖSSÄ

Ohjelmistotestien toteuttaminen eri testausmenetelmillä on hyvä tapa varmistaa ohjelmiston uusien toiminnallisuuksien toimiminen halutulla tavalla ja minimoida ohjelmistovirheet. On tärkeää testata myös aiemmin toteutettujen toimintojen säilyminen oikeanlaisena ohjelmiston kehittyessä sekä varmistaa ohjelmiston ylläpidettävyys ja jatkokehitysmahdollisuudet. Eri testausmenetelmillä on omat käyttötarkoituksensa ja menetelmät eivät sulje pois toisiaan. Koska testeihin ja muuhunkin ohjelmistotuotantoon varatut resurssit ovat aina rajalliset, laadun varmistaminen pitää tehdä valikoiden ja suunnitellen testit huolella, jotta resurssit kohdennettaisiin sellaisiin testitapauksiin, joista on eniten hyötyä ohjelmistovirheiden ennaltaehkäisemisessä sekä löytymisessä. Testaus on laadukkaan ohjelmistotuotannon yksi kulmakivistä. Mitä aikaisemmassa vaiheessa ohjelman virhetoiminto huomataan, sen helpompaa virhe on korjata. Jos ohjelmavirhe pääsee tuotantoon asti, asiakastyytyväisyys kärsii ja virheen kustannukset nousevat.

Testikäytännöt vaihtelevat eri ohjelmistoyrityksissä ja -projekteissa. Ohjelmistokehittäjän vastuulla on kirjoittaa yksikkötestit, jotka testaavat tarkasti rajatusti vain niitä toimintoja, jotka on juuri toteutettu. Projekteissa ja yrityksissä voi olla erikseen testaajia, joiden tehtävänä on testata uudet toiminnallisuudet suunnittelemalla ja toteuttamalla automaatio- ja manuaalitestejä. Näiden testien avulla toiminnallisuuksia sekä eri toimintojen yhteensopivuutta testataan laajemmin. Joissain yrityksissä on käytössä myös tapoja, jossa testausvastuu on asiakkaalla.

Automaatio- ja manuaalitestauksessa todennetaan, toimivatko ohjelmiston osat yhteen eli puhutaan integraatiotestauksesta. Automaatiotestaus ajetaan nimensä mukaisesti koneellisesti ilman ohjelmistotestaajan ajonaikaisia toimenpiteitä. Tarkastellaan automaatiotestiesimerkkinä tähän päättötyöhön liittyvää PLP-ohjelmistotuotetta, jolla hallitaan vakuutussopimuksia. Esimerkkitestissä halutaan testata vakuutussopimuksen perustaminen tietyillä henkilö- ja sopimustiedoilla loppukäyttäjän käyttöliittymän kautta. Testi etsii lomakkeelta ne kentät, joiden nimet täsmäävät testin määrittelyyn. Testi täyttää kyseiset kentät ennalta määrätyillä tiedoilla ja tallentaa lomakkeen testissä määritellyn tallennusnapin avulla. Tietojen syöttämisen jälkeen testissä luetaan tietokannasta tulokset: löytyvätkö sieltä vastaavat tietueet eli ovatko tiedot tallentuneet oikein tietokantaan? Vastaako ohjelman palauttama ilmoitus onnistuneesta tietojensyötöstä testin olettamaa merkkijonoa? Mikäli tiedot täsmäävät, testi on mennyt läpi.

Manuaalitestauksessa oikea ihminen, yleensä testaaja, suorittaa ennakolta suunnitellut testit käyttöliittymän avulla. Jos kyseessä on vastaavan tyyppinen vakuutussopimuksen luonti kuin automaatiotestiesimerkissä, testaaja syöttää tiedot käyttöliittymän lomaketta

käyttäen, etsien silmämääräisesti ne kentät, joihin tiedot syötetään. Testaaja tallentaa lomakkeen tietoineen ja tarkistaa, että ohjelmisto antaa toivotun ilmoituksen tietojen syöttämisen onnistumisesta. Testaaja käy vielä tarkistamassa sopimusrekisteristä, löytyykö testin aikana luotu sopimus sieltä ja vastaako sopimuksen sisältö testaajan olettamaa sisältöä.

Tässä päättötyössä keskitytään tarkastelemaan ohjelmiston laatua ylläpidettävyystekijöiden ja ohjelmistokehittäjän näkökulmasta, jonka vuoksi tarkastelun keskiössä ovat yksikkötestausmenetelmät. Yksikkötestit testaavat huomattavasti pienempiä kokonaisuuksia kuin integraatiotestit. Yksikkötestin kohteena voi olla ohjelmistoluokka tai luokan sisällä oleva yksittäinen metodi. Perinteisesti ohjelmistokehityksen toteutusvaihe noudattaa Test-Last Development (TLD) -kehitystapaa, jossa ohjelmistokehittäjä tutustuu ensin määrittelydokumentaatioon ja toteuttaa sitten määrittelyn mukaisesti ohjelmaan jonkin toiminnallisuuden. Tämän jälkeen syntyneelle toiminnolle suunnitellaan yksikkötestit ja uudet osat testataan näiden testien avulla. Yksikkötestien toteutus voidaan toteuttaa myös käänteisellä tavalla eli toteuttamalla yksikkötestit ennen ohjelmakoodin toteuttamista, jolloin kehitystapana on testivetoinen ohjelmistokehitys (Test-Driven Development, TDD). Verrattuna TLD-kehitystapaan, TDD-kehitystavalla saavutetaan monia etuja liittyen ohjelmiston laatuun ja kehittäjän työskentelyyn. TDD-kehitystavan työskentelymetodiikkaan ja kehitysmenetelmän etuihin tutustutaan tarkemmin luvussa 2.5 *Testivetoinen ohjelmistokehitys*.

Kuten automaatiotestit, myös yksikkötestit on suunniteltu pysyväksi testaustavaksi, eli testit säilyvät ohjelmistokokonaisuudessa ohjelmiston kehittyessä. Ohjelmistokehittäjä suorittaa kaikki ohjelmistoon toteutetut yksikkötestit kehitystyönsä aikana tai ainakin viimeistään uuden toiminnallisuuden valmistuttua. Automaatiotestit todentavat toiminnallisuuden ohjelmistokokonaisuudessa, kun uusi toiminnallisuus liitetään osaksi kokonaissovellusta. Manuaalitestaus vaatii testaajan työpanosta ja näiden testien suorittaminen on enemmän tapauskohtaista ja kertaluonteista.

Testien suunnittelu ja toteuttaminen voi jäädä vähäiselle eri syistä. Jos ohjelmiston tilaajaa ei saada vakuutettua testauksen tarpeellisuudesta, tilaaja ei ehkä ole valmis maksamaan testien toteuttamisesta. Aikataulukiireet voivat omalta osaltaan aiheuttaa paineita julkaista uusia ominaisuuksia, jolloin testien toteutuksesta tingitään. Myös ohjelmistokehittäjän omat yksilölliset taipumukset ohjaavat yksikkötestien toteuttamista. Jos kehittäjä arvottaa testien merkityksen korkealle, kehittäjä toteuttaa yksikkötestejä pitkässä juoksussa enemmän kuin kehittäjä, jonka mielestä ohjelmistokoodi on valmis, kun toiminnallisuuden suorittava koodiosuus on valmis. Yrityksen omat toimintatavat ja linjaukset yksikkötestien toteutuksesta ja määristä tasoittavat kehittäjien yksilöllisiä eroja. Hyvä toimintatapa on käyttää koodin katselmoinnissa henkilöitä, joilla on yrityksen linjan mukainen näkemys yksikkötestien toteutuksen riittävyydestä. Mikäli yksikkötestien määrä tai laatu ei ole riittävä, katselmoija palauttaa ohjelmakoodin takaisin ohjelmistokehittäjälle täydennettäväksi.

2.1 Yksikkötestaus

Yksikkötestien suunnittelu ja toteutus on ohjelmistokehittäjän työtä. Yksikkötesteissä voidaan syöttää ohjelmalle myös poikkeuksellisia arvoja, joita ohjelma ei kykene suorittamaan. Tilanteessa testataan, että ohjelman toiminto käsittelee virhetilanteen toivotulla tavalla ja että ohjelma toipuu virheestä hallitusti. Esimerkkinä yksikkötestistä voidaan tarkastella tilannetta, jossa ohjelman toivotaan laskevan kaksi aliohjelmalle parametrina annettavaa lukua yhteen. Mikäli parametreina annetaankin lukujen sijasta tekstimuotoinen syöte, aliohjelman pitäisi suoriutua tästä hallitusti ilman, että koko ohjelman suoritus häiriintyy. Esimerkin yksikkötestin toteutuksessa syötetään aliohjelmalle tekstimuotoiset parametrit ja verrataan aliohjelman palauttamaa tulosta toivottuun virheilmoitukseen.

Usein ohjelmistokehitys noudattaa mallia, jossa ohjelmistokehittäjä kirjoittaa toteutettuun toiminnallisuuteen liittyvät yksikkötestit, kun kehitettävä toiminnallisuus tai sen osa on valmis (vrt. 2.5 Testivetoinen ohjelmistokehitys). Ohjelmakoodin tekijä myös suorittaa yksikkötestit ja varmistaa, että ne menevät läpi. Kaikki ohjelmistoon liittyvät, myös aiemmin toteutetut yksikkötestit suoritetaan ohjelmakoodin kääntämisen yhteydessä. Ohjelmakoodin kääntämisellä tarkoitetaan ohjelmakoodin muutamista konekieliseen muotoon, jolloin ohjelmakoodi muuntuu tietokoneen ymmärtämäksi suoritettavaksi ohjelmistoksi.

Regressiotestaus tarkoittaa sitä, kun aiemmin toteutetut yksikkötestit säilyvät ohjelmistokokonaisuudessa ja ne ajetaan aina ennen uuden toiminnon hyväksymistä (Leung, 1989). Regressiotestauksen yhteydessä huomataan, jos uusi toiminnallisuus aiheuttaa virheen johonkin aiemmin toteutettuun toiminnallisuuteen. Yksikkötestien kirjoittaminen osana ohjelmistotuotantoprosessia tuo turvaa ohjelmistokehittäjän työhön, kun uusien toimintojen valmistuttua eri ohjelmistokehitysvaiheissa toteutettujen yksikkötestien suorittamisella varmennetaan, että aiemmat toiminnallisuudet toimivat edelleen oikein. Yksikkötestejä voidaan myöhemmin käyttää kuvauksena ja apuna selvitettäessä, miten jokin ohjelman osa tai aliluokka toimii. Kun testitapauksia tutkii, selviää, miten itse ohjelmaa voidaan käyttää ja mitä arvoja aliohjelma palauttaa.

Kukin yksikkötesti toteutetaan testaamaan pientä ohjelmiston osaa eli yhtä luokaa tai aliluokkaa. Yksikkötesti vertaa ohjelman palauttamaa arvoa testissä asetettavaan arvoon. Mikäli ohjelman palauttama arvo on sama kuin yksikkötesti olettaa, testi on onnistunut (*passed*). Jos arvot eivät täsmää, testi epäonnistuu (*failed*). Testit suunnitellaan ajettavaksi automaattisesti napin painalluksella, jotta testit on helppo toistaa tarvittaessa. Ohjelmistokokonaisuuteen liittyviä yksikkötestejä pitää voida ajaa samanaikaisesti ilman, että ne häiritsevät toisiaan tai että testin suorituksen tulos riippuu jonkin toisen testin suorittamisesta. Koska yksikkötestillä pyritään testaamaan vain testauksen kohteena olevaa ohjelman toimintoa, yksikkötestit pyritään toteuttamaan siten, että ne eivät sisällä riippuvuuksia muihin ohjelmiston sisäisiin tai ulkoisiin osiin, käytä apunaan tietokantaa, kommunikoi verkkoyhteyden avulla eivätkä lue tai kirjoita tietoja tiedostojärjestelmään (Sprengers,

2017). Jos riippuvuuksia ei pystytä poissulkemaan, saatetaan joutua tilanteeseen, jossa riippuvuudessa tai monimutkaisemmassa usean riippuvuuden ketjussa ilmenee ongelma ja näin ollen myös riippuvuuden sisältävä yksikkötesti epäonnistuu. Tämä aiheuttaa ongelmia virheen jäljityksessä, kun testin epäonnistuminen johdattaa virheen korjaajan tutkimaan ongelmaan liittymätöntä ohjelmakoodin osaa. Ongelmaa ryhdytään etsimään aivan muualta kuin missä oikea virhe aiheutuu. Yksikkötestauksessa riippuvuuksia sisältävät ohjelmiston osat korvataan toteuttamalla mock-objekteja (*mockups, mock-ups, mock objects*), jotka ikään kuin näyttelevät oikeita objekteja (ks. 2.3 Mock-objektit). Objektien simuloinnilla tavoitellaan tilaa, jossa mock-objekti käyttäytyy kuten todellinen objekti.

2.2 Yksikkötestien asettaminen ja annotaatiot

Eri ohjelmointikielille löytyy runsaasti vaihtoehtoisia yksikkötestikehyksiä, joiden avulla testit toteutetaan. Wikipediassa pelkästään Java-ohjelmointikielelle eri yksikkötestikehyksiä on listattu noin 50 (Wikipedia, 2018). Näistä JUnit ja TestNG ovat työn tilaajan yleisesti käytössä olevat yksikkötestialustat Java-ohjelmointikielellä toteutetuissa projekteissa. Tämän päättötyön yksikkötestien tarkastelussa, esimerkeissä ja toteutuksissa käytetään TestNG-kehystä.

Yleensä yksikkötestiluokat sijaitsevat samassa ohjelmistoprojektissa muiden java-luokkien kanssa. Hakemistorakenteessa Java-luokat sijoitetaan tyypillisesti src/main/java- ja testiluokat src/test/java-hakemistorakenteen alle. Hakemistorakenne ei kuitenkaan ole määrittävä tekijä testien ja ohjelmakoodiluokkien välillä. Mikä tahansa java-luokan metodi voi toimia yksikkötestin suorittavana metodina, kun luokan *public* -tyyppinen metodi merkitään *@Test* –annotaatiolla.

Monesti testin suorittaminen tarvitsee alustavia toimenpiteitä ennen testien suorittamista tai testien halutaan alkavan tietystä tilanteesta. Yksikkötestikehyksessä käytetään eri annotaatioita varsinaisille testeille ja testien tilan alustuksille. Oheisessa listauksessa on esitelty valikoituja annotaatioita, joita käytetään jäljempänä päättötyön käytännön esimerkeissä (ks. 4 Rajapinnan toteutus, testaus ja testitulokset) ja jotka ovat oleellisia päättötyön kannalta (Beust, 2018):

- *@BeforeClass*-annotaatiolla merkitty metodi suoritetaan kerran testiluokan suorituksen yhteydessä, ennen muita testimetodeita tai niiden alustuksia.
- *@BeforeMethod*-annotaatiolla merkitty metodi ajetaan aina ennen jokaista *@Test* merkittyä testimetodia, riippumatta siitä, missä testiluokan kohdassa kyseinen metodi sijaitsee. Jokainen luokan testi alkaa samasta tilasta tässä metodissa esiteltyjen muuttujien ja arvojen osalta.
- Ohjelmaluokka voi sisältää useamman *@Test*-annotaatiolla merkityn suoritettavan testimetodin. Jos testiluokassa on *@BeforeTest*-annotaatiolla merkitty metodi, se ajetaan juuri ennen metodin jäljessä seuraavaa *@Test*-merkittyä testime-

todia, joten tämän annotaation sijainnilla testiluokassa on merkitystä. *@BeforeTest*-annotaatiolla merkitty metodi liittyy tiukasti jäljessä seuraavaan testimetodiin.

Yksikkötestin suunnittelun lähtökohdaksi on asetettu, että yksikkötestejä voi ajaa missä järjestyksessä tahansa ja testin tulos ei saa riippua muista suoritettavista yksikkötestiluokista tai luokan sisällä olevista testimetodeista. Muuttujien tilat ja arvot saattavat muuttua testin ajon yhteydessä, joten testin suunnittelijan pitää olla tarkkana, ettei joku toinen testimetodi saa käyttöönsä toisen testin suorituksessa muuttuneita arvoja. Myös testejä ennen suoritettavia alustuksia pitää tarkastella. Esimerkiksi *@BeforeTest*-annotaatiolla merkittyjä arvoja ja muuttujia ei palauteta alkutilaansa, kuten ne palautetaan *@BeforeMethod*-merkityn metodin arvojen kohdalla. Ohjelmassa 1 esitellään testiannotaatioiden käyttöä ja testataan kuvitteellista laskinta. Laskimessa oletetaan olevan metodi kahden luvun yhteenlaskemiseksi sekä toinen metodi kahden luvun jakamiseksi keskenään.

```
package com.websystique.testng;

import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class TestNGMethodsExample {

    @BeforeMethod
    public void beforeMethod() {
        System.out.println("beforeMethod()");
    }

    @AfterMethod
    public void afterMethod() {
        System.out.println("afterMethod()\n");
    }

    @Test
    public void testAdd() {
        System.out.println("testAdd()");
        Calculator calculator = new Calculator();
        Assert.assertEquals(calculator.add(15, 2) , 17);
    }

    @Test
    public void testDivide() {
        System.out.println("testDivide()");
        Calculator calculator = new Calculator();
        Assert.assertEquals(calculator.divide(16, 0), 16);
    }

}
```

Ohjelma 1. *Esimerkkiohjelma testiannotaatioiden ja Assert-luokan metodien käytöstä. Esimerkin ohjelmakoodia on hieman yksinkertaistettu lähteen esimerkistä (WebSystique, 2017)*

Assert-luokkaa käytetään testien ajossa testituloksen oikeellisuuden tarkastelussa. Luokan metodit ovat ikään kuin väittämiä, jossa testataan väitteen totuusarvoa. Mikäli väite on tosi (*true*), testi on onnistunut. Väittämän ollessa epätosi (*false*), testitapaus on epäonnistunut. Ohjelmassa 1 käytetään esimerkkinä Assert-luokan metodia assertEquals, jossa ensimmäisenä parametrina toteutetaan suoritettava testi ja verrataan testin palauttamaa arvoa toisena parametrina annettavaan arvoon. AssertEquals-metodi olettaa ensimmäisen ja toisen parametrin olevan keskenään samanlaiset ja muuttujientyyppien olevan keskenään samat (esimerkiksi String eli merkkijono tai double eli kaksinkertaisen tarkkuuden desimaaliluku). Esimerkkitestissä testattavan Calculator-luokan yhteenlaskun suorittavalle add()-metodille annetaan parametreina kaksi lukua, 15 ja 2. Yhteenlaskun tulosta verrataan metodin palauttamaan testissä asetettuun arvoon 17. Jakolaskun toimittavalle divide()-metodille annetaan parametreina luvut 16 ja 0 ja testataan metodin palauttamaa tulosta lukuarvoon 16. Kun Ohjelman 1 testit suoritetaan, testitulokseksi saadaan Kuvassa 1 esitetty testitulos.

```
beforeMethod()
testAdd()
afterMethod()

beforeMethod()
testDivide()
afterMethod()

PASSED: testAdd
FAILED: testDivide
java.lang.ArithmeticException: / by zero
    at com.websystique.testng.Calculator.divide(Calculator.java:10)
```

Kuva 1. *Testiohjelman (Ohjelma 1) suorituksen tulos eli testin tulos (WebSystique, 2017)*

Kaikista esitellyistä annotaatioista on olemassa vastaavat annotaatiot @After-alkuliitteisinä. Nimensä mukaisesti metodi suoritetaan annotaation loppuosassa määritellyn vaiheen jälkeen. Esimerkiksi *@BeforeClass*-annotaation avulla testien alussa asetetut ohjelmistokonfiguraatiot ja muuttujat voidaan asettaa alkuperäiseen tilaansa *@AfterClass*-annotaatiolla merkityssä metodissa. *@After*-alkuisilla annotaatioilla huolehditaan muuttujien ja testiympäristön palauttamisesta alkutilaansa, jotta seuraava testi voidaan suorittaa itsenäisesti riippumattomana muista yksikkötestien suorituksista.

Ohjelmassa 1 suoritetaan ennen varsinaisia testejä annotaatioilla merkittyjä alustavia toimenpiteitä. Ohjelmassa jokaisen testiannotaatioihin liittyvän metodin toteutukseen on laitettu tulostumaan testimetodin nimi testin suoritusjärjestyksen havainnollistamiseksi. Kuvan 1 testituloksessa nähdään, että *@BeforeMethod*-annotaatiolla merkityt metodit suoritetaan aina ennen jokaista *@Test*-merkinnällä varustettua testitapausta. Vastaavasti *@AfterMethod*-annotaatiolla merkityt metodit suoritetaan jokaisen testin jälkeen.

Varsinaisessa testiosuudessa, eli *@Test*-annotaatiolla merkityssä testAdd()-luokassa suoritettava kahden luvun yhteenlaskeva add()-metodi toimii oikein. Koska add()-metodin palauttama arvo vastaa testissä asetettua arvoa 17, testituloksessa kyseisen metodin suorituksen yhteydessä on palautunut arvo PASSED. Testinsuorituksessa toisena testinä suoritetaan testDivide()-metodi, jossa yritetään jakaa luku 16 nollalla ja jakotulokseksi testissä oletetaan tulevan luku 16. Oletettavasti testi epäonnistuu, koska jakolaskun tulos ei ole 16 eikä nollalla voi ylipäänsä jakaa. Tässä esimerkissä on haluttu näyttää tilanne, jossa testi epäonnistuu. Jakolaskun suorittavassa testissä kannattaisikin testata nollalla jaettaessa, että testattavan Calculator-luokan divide()-metodi palauttaa AritmethicException-tyyppisen poikkeuksen ja huolehtia divide()-metodin todellisessa toteutuksessa siitä, että nollalla jaettaessa poikkeuksesta suoriudutaan hallitusti.

Ohjelmassa 1 esiteltiin Assert-luokan testimetodi assertEquals(). Luokka sisältää myös muita metodeja testien suoritukseen. Esimerkiksi assertTrue() ja assertFalse() -metodeille annetaan parametrina ainoastaan testattava metodi ja testitulosta verrataan totuusarvoon tosi (true) assertTrue() metodin yhteydessä. Vastaavasti testituloksen odotetaan olevan totuusarvoltaan epätosi (false) assertFalse() metodin yhteydessä. Calculator-luokan isInteger()-metodin palauttamaa totuusarvoa voidaan testata esimerkiksi testeillä

```
Assert.assertTrue(calculator.isInteger(“16”));
Assert.assertFalse(calculator.isInteger(“Tekstiä”));
```

Esimerkin isInteger()-metodi ottaa parametrinä String-tyyppisen muuttujan, ja koittaa muuntaa sen int-tyyppiseksi. Mikäli tyyppimuunnos onnistuu, metodi palauttaa tosi-totuusarvon. Muussa tapauksessa metodi palauttaa arvon epätosi. Molemmat esimerkin testit onnistuvat, koska ensimmäisessä tapauksessa testituloksen oletetaan olevan tosi ja toisessa epätosi.

2.3 Mock-objektit

Todellisessa elämässä ohjelmistot ovat huomattavasti monimutkaisempia kuin Ohjelmassa 1 testattu lukujen yhteenlasku- ja jakometodit. Metodien sisään voi olla upotettuna toiminnallisuuksia, jotka hyödyntävät ohjelmiston sisäisiä tai riippuvuuksien kautta tulevia ulkoisia komponentteja. Metodin ulkopuoliset komponentit saattavat sisältää taas lisää riippuvuuksia. Thomas *et al.* kehottavatkin olemaan varovainen riippuvuuksien käytössä yksikkötestauksessa (Thomas *et al.* 2002). Jos testattavan metodin kaikki riippuvuudet alustetaan testin käytettäväksi, joudutaan tilanteeseen, jossa huomattava osa ohjelmiston komponenteista on käytössä yhden yksikkötestin suorituksessa. Silloin yksikkötesti ei enää testaakaan vain kohteena olevaa metodia, vaan koko riippuvuuksien kirjoa. Tällaisessa tapauksessa jotain ohjelmiston osan toiminnallisuutta muutettaessa saattaakin jokin täysin muuta ohjelmiston osaa testaava yksikkötesti epäonnistua, koska testi tarvit-

see onnistuakseen muutettua ohjelmiston osaa riippuvuuksien kautta. Ongelman jäljittäminen voi siinä tapauksessa olla aikaa vievää, kun yksikkötestin epäonnistuminen johtaa ongelman selvittämisessä aivan muualle, kuin missä oikea ongelma aiheutuu.

Testattavan ohjelmistokoodin riippuvuuksia ja suorituksessa tarvittavia tietokantayhteyksiä voidaan jäljitellä mock-objekteilla. Tim Mackinnon, Steve Freeman ja Philip Craig ovat kehittäneet mock-objektit sekä Java-ohjelmointiympäristön mock-objektikehyksen (Thomas *et al,* 2002). Mock-objektien avulla yksikkötestit voidaan suorittaa eristyksessä muusta ohjelmakoodista. Mackinnon *et al.* käyvät artikkelissaan läpi tilanteita ja syitä, miksi ohjelmistoriippuvuudet kannattaa korvata mock-objekteilla, jotka ikään kuin näyttelevät oikeita objekteja (Mackinnon *et al.,* 2001):

- Lykätään infrastruktuuriin sitoutumista niin pitkälle kuin mahdollista. Kun ohjelmistoa toteutetaan, infrastruktuurin valintaa ei välttämättä vielä ole tehty. Voi olla esimerkiksi, että mitään tiettyä tietokantaa ei ole vielä valittu, jolloin mock-objektit toteuttavat tietokannan toimintoja.
- Todellinen objekti on hankala alustaa käytettäväksi.
- Todellisen objektin käyttäytyminen on hankala saada testin suorituksessa aikaiseksi, kuten esimerkiksi tietoverkossa ilmenevät ongelmat.
- Todellinen objekti on hidas.
- Todelliseen objektiin liittyy loppukäyttäjän käyttöliittymä.
- Kun testin pitää selvittää todelliselta objektilta, miten sitä käytetään.
- Todellista objektia ei vielä ole olemassa tai objekti on toteutettu toiseen ohjelmistoon.

Aiemmin tässä päättötyössä tarkastelimme vakuutussopimuksen luontia ja siihen liittyvien manuaalisten ja automaatiolla suoritettavien integraatiotestien toteuttamista. Tutkitaan tässä yhteydessä testitapausta yksikkötestauksen ja mock-objektien näkökulmasta. Yksikkötesti testaa, onnistuuko vakuutussopimuksen luonti käyttämällä suoraan hyödyksi ohjelmistoluokkia. Jotta sopimus voidaan luoda, pitää sopimukseen liittää asiakas. Todellista asiakasobjektia hyödyntävän yksikkötestin tulisi ensin perustaa asiakas kaikkine vaadittavine asiakastietoineen tai käyttää hyödykseen jo olemassa olevaa asiakasrekisteriä. Asiakkaan perustaminen on työläs prosessi, joka pitäisi suorittaa vain sitä varten, että asiakkaan yksilöivä id-numero saadaan liitettyä luotavaan sopimukseen. Asiakas-objektia saatetaan myöhemmin muuttaa esimerkiksi siten, että suomalaisen henkilötunnuksen validointi muuttuu hyväksymään vain ruotsalaisia henkilötunnuksia. Muutoksen jälkeen vakuutussopimuksen luonti yksikkötestissä epäonnistuu, jos testin aikana luodaan uusi asiakas aiemmalla henkilötunnuskäytännöllä. Jos sopimusta perustettaessa asiakastieto haetaan olemassa olevasta asiakasrekisteristä, se aiheuttaa omat ongelmansa. Tietokantayhteyksiä pitäisi yksikkötestauksessa välttää ja asiakastiedot ovat tallennettuna tietokantaan. Myös asiakasrekisterin sisältö muuttuu ajan myötä. Jonain ajanhetkenä testissä

käytettävä asiakas ei ehkä olekaan enää olemassa rekisterissä tai asiakastiedot ovat muuttuneet. Yksikkötesti pitäisi kuitenkin olla toistettavissa milloin tahansa ja testitulos pitäisi pysyä samana.

Asiakasobjekti voitaisiin toteuttaa mock-objektina vakuutussopimuksen luonnin testauksessa Ohjelmassa 2 esitetyllä tavalla. Mock-objektien toteuttamisessa eli mocking-kehyksenä käytetään Ohjelman 2 esimerkissä Mockito-testikehystä, joka on avoimen lähdekoodin testikehys ja jota käytetään myös päättötyön varsinaisessa ohjelmistototeutuksessa. Mock-objektina toteutetun luokan ohjelman suorituksen kannalta oleelliset metodit korvataan when(class.method).thenReturn(value) -tyyppisillä komennoilla, joilla asetetaan luokan oikea metodi palauttamaan haluttu arvo (value).

```
import static org.mockito.Mockito.*;
....
private Policy policy;
private Client client;

@BeforeClass
public void init(){
      client = mock(Client.class);
      when(client.getClientNumber().thenReturn(12345);
}

@Test
public void testCreatePolicyForPerson(){
      when(client.isClientPerson().thenReturn(True);
      createPolicy();
      assertEquals(policy.getClientNumber(), 12345);
      assertEquals(policy.getTaxCode(), 24);

}

@Test
public void testCreatePolicyForCompany(){
      when(client.isClientPerson().thenReturn(False);
      createPolicy();
      assertNull(policy.getTaxCode());
}
....
public void createPolicy() {
      policy = new Policy();
      policy.setProductId(2000);
      ....
      if (client.isClientPerson()) {
            policy.setTaxCode(24);
      }
      policy.setClientNumber(client.getClientNumber());
}
```

Ohjelma 2. *Ohjelmakoodiotos kuvitteellisesta testistä, jossa asiakasobjekti toteutetaan mock-objektina.*

Ohjelma 2 sisältää kaksi testiä, joissa toisessa luodaan sopimus henkilölle, toisessa asiakkaana on yritys. Molemmissa testeissä luodaan uusi sopimus createPolicy()-metodin avulla. Henkilöasiakkaan ollessa kyseessä sopimukseen tallennetaan verokoodi, yrityksen tapauksessa verokoodin tulisi jäädä ilman arvoa. Ohjelmassa mock-objekti näyttelee Client-luokkaa. Luokalla on metodi getClientNumber(), joka asetetaan testissä palauttamaan asiakasnumeron 12345. Metodi isClientPerson() asetetaan palauttamaan totuusarvon tosi henkilöasiakkaan ollessa kyseessä, yritysasiakkaan kohdalla palautetaan totuusarvo epätosi. Kun testataan sopimuksen luontia henkilöasiakkaalle, assertEquals()-metodi testaa, vastaako asiakasnumero ja verokoodi testissä asetettuja lukutyyppisiä arvoja. Yritysasiakaan sopimuksessa assertNull()-metodi testaa, että verokoodi on tyhjä (Null).

2.4 Ohjelmiston lähdekoodin refaktorointi

Refaktoroinnilla tarkoitetaan ohjelmakoodin parantamista. Toiminnan päämääränä on säilyttää ohjelman ulkoinen toiminta samalla, kun koodin sisäistä rakennetta muutetaan pienin muutoksin ja askelin suuntaan, jossa koodin luettavuus, ylläpidettävyys ja laajennettavuus paranee (Acharya, 2013, s.36). Ohjelmiston jatkokehityksen kannalta on tärkeää, että ohjelman rakenne on selkeä ja että siitä voi päätellä, mitä ohjelman tulisi tehdä. Enzlerin puhtaan koodin tuottamisen oppaasta selviää, minkälaiseen ohjelmakoodirakenteeseen ja nimeämisiin TDD-kehitystyöskentelyssä tulisi pyrkiä (Enzler, 2014). Opas on hyödyllinen myös muilla kehitystavoilla työskenteleville, koska puhtaan koodin periaatteet pätevät myös niihin.

Ohjelmiston elinkaaren aikana projekteissa vaihtuvat tekijät ja vaikka kyseessä olisikin oma kooditoteutus, sen yksityiskohtia ja erityistoiminnallisuuksia on vaikea muistaa pitkän ajan kuluttua. Jos toteutuksessa on tyydytty tasoon, jossa koodi ainoastaan toteuttaa toiminnallisuuden sen enempää puuttumatta ohjelmakoodin rakenteeseen, ensimmäisessä toteutusvaiheessa säästetty työmäärä aiheuttaa kustannuksia ohjelmiston jatkokehitysvaiheessa. Ohjelmistokehittäjät ryhtyvät tällaisten epäselkeiden ohjelmakoodien muokkaustehtäviin yleensä vastahakoisesti. Kun ohjelma on vaikeaselkoinen, muokkaus saattaa aiheuttaa uusia ongelmia ja virheitä.

Mikäli myös yksikkötestit on toteutettu puutteellisesti, ohjelmistokehityksen riskit kasvavat entisestään. Jos jatkokehitettävään ohjelmistoon on tunnollisesti toteutettu yksikkötestit, ne pitävät huolta, että ohjelmakoodi toimii ulospäin samalla tavalla refaktoroinnin aikana ja koodin sisäisen rakenteenmuokkauksen päätyttyä. Ohjelmiston rakenteita voi muuttaa aika radikaalistikin, koska yksikkötestit ilmentävät nopeasti, jos muokkaustyön aikana ohjelman toiminnallisuus muuttuu.

Refaktorointia tehdään ohjelmistokehityksen eri vaiheissa. Milloin ohjelmoija tietää, että on aika aloittaa refaktorointi? Acharyan mukaan seuraavat kolme vaihetta antavat selkeän alkupisteen refaktoroinnin aloittamiselle (Acharya, 2013, s.37):

- Uuden ominaisuuden lisääminen on hankalaa aiemman kooditoteutuksen monimutkaisuuden vuoksi. Aiempaa toteutusta pitää refaktoroida ennen uusien ominaisuuksien lisäämistä.
- Ohjelmakoodista löytyneen virheen eli bugikorjauksen yhteydessä.
- Koodikatselmoinnin aikana.

Kun toteutetaan uusi ominaisuus olemassa olevaan ohjelmistokoodiin, ohjelmistokehittäjän pitää ymmärtää olemassa olevan koodin toiminnallisuus. Mikäli toiminnallisuuden toteutus on niin monimutkainen, että siihen on hankala ryhtyä lisäämään uusia ominaisuuksia, on tarpeen ensin refaktoroida aiempaa toteutusta. Myös ohjelmistossa esiintyvän virheen korjauksen yhteydessä voi refaktoroida.

Uuden ominaisuuden toteutuksen yhteydessä on luontevaa tehdä refaktorointia syntyvälle ohjelmakoodille. Ohjelmistokehittäjän omat työskentelymetodit usein määräävät, missä vaiheessa refaktorointityötä suoritetaan ohjelmointityön aikana. Koodin parannuksia voi tehdä jatkuvasti aina pienten osatoteutusten valmistuttua ja jossain kohdassa metodit voivat vaatia isompaa koodirakenteen selkeyttämistä. Noudatettaessa testivetoista ohjelmistokehitysmenetelmää, ohjelmointityö noudattaa menetelmän omaa kehityssykliä, jossa refaktoroinnilla on oma paikkansa. Käytännön työskentelyssä ohjelmistokehittäjä kuitenkin itse ratkaisee, kuinka paljon refaktorointia tehdään kehityksen alla olevaan ohjelmaosioon. Isommat rakennemuutokset saattavat koskettaa eri kehityssykleissä syntynyttä koodia, joten tällaiset refaktorointitapaukset eivät välttämättä noudata TDD-syklin vaiheita orjallisesti.

Lopullinen refaktorointi suoritetaan koodikatselmoinnin yhteydessä. Ohjelmistokehittäjä saattaa sokeutua omalle toteutukselleen eikä oivalla otollisia parantamiskohteita. Kun toinen ohjelmistokehittäjä katsoo toteutusta, hän saattaa huomata toteutuksia, jotka voisi esittää yksinkertaisemmalla ehtolauseella tai luokkien, metodien ja muuttujien nimeämisissä on epäloogisuutta. Suoritetaan refaktorointi missä vaiheessa tahansa, jossain pisteessä koodiin pitää olla tyytyväinen ja hyväksyä, että koodi on nyt riittävän hyvä. Aina löytyy jotain parannettavaa, mutta kooditoteutuksen hiomista ei voi jatkaa loputtomiin.

Enzel esittelee Clean ATDD/TDD Cheat Sheet -artikkelissaan yksityiskohtaisesti, mihin muotoon refaktorointityöskentelyssä koodin rakenteita ja nimeämisiä pitäisi pyrkiä viemään (Enzel, 2014). Tämän päättötyön kirjallisessa osuudessa ei paneuduta refaktoroinnin yksityiskohtiin kovin syvällisesti, vaikka artikkelissa esitettyjä refaktoroinnin periaatteita onkin noudatettu ohjelmointityöskentelyssä. Acharya mainitsee hieman yleisemmällä tasolla kolme refaktoroinnin tavoitetta, joihin pitäisi pyrkiä ohjelman rakenteen parantamisessa (Acharya, 2013, s.36). Näistä tavoitteista käytetään nimityksiä DRY, Open/Closed ja YAGNI. Ohjelmakoodissa saman asian toteuttavat osat eli koodiduplikaatit tulisi poistaa (DRY, Don’t Repeat Yourself). Jatkokehityksessä tai virheenkorjauksessa joudutaan muuten tilanteeseen, jossa koodimuutos joudutaan tekemään useampaan samansisältöiseen ohjelmaosuuteen. Open/Closed tarkoittaa, että ohjelmakoodin tulisi

olla avoin laajennuksille, mutta muokkauksilta suljettu. Kun uusi toiminnallisuus lisätään, sen tulisi aiheuttaa mahdollisimman vähän muutoksia olemassa olevaan ohjelmakoodiin. YAGNI:lla (You Aren't Gonna Need It) pyritään välttämään ylisuunnittelua. Koodi tulisi toteuttaa tämänhetkiseen tarpeeseen eikä toteuttaa sellaisia osia, joita ehkä tulevaisuudessa tarvitaan.

2.5 Testivetoinen ohjelmistokehitys

Testivetoinen ohjelmistokehitys tarkoittaa ohjelmistokehityksessä tapaa, jossa yksikkötestit kirjoitetaan ennen ohjelmakoodin toteuttamista. Toimintatapa ohjaa kehittäjää toteuttamaan toiminnallisuuden määrittelyiden mukaiseksi, koska hän joutuu tutustumaan tarkasti määrittelyyn ennen varsinaisen ohjelmointityön aloittamista. Testivetoisen ohjelmistokehitysmenetelmän kehittäjänä pidetään Yhdysvalloista kotoisin olevaa Kent Beckiä (Sujoy, 2014).

Kun ohjelmistokehityksessä käytetään TDD -menetelmää, pilkotaan toteutettava toiminnallisuus ensin pieniin osiin ja testin suunnittelussa keskitytään testaamaan vain yhtä tällaista pientä ohjelman osaa. Testivetoinen ohjelmistokehitys on kehitystapa, joka ohjaa myös toteutettavan ohjelmiston rakennetta yksikkötestikelpoiseksi. Tavassa, jossa yksikkötestit toteutetaan vasta jälkikäteen, kehittäjä saattaa joutua muuttamaan toimintojen rakenteita. Koska ohjelmiston rakennetta ei ole mietitty testien kannalta ohjelmiston toteutusvaiheessa, osa toiminnoista voi olla hankalasti tai ei ollenkaan testattavissa yksikkötestausmenetelmillä.

Testien suunnittelu ennen toteutusta ohjaa ohjelmistotuotantoa parempaan laatuun ja testien toteutus ei jää ominaisuuden valmistumisen jälkeen irralliseksi, omaksi toiminnokseen. Työn edetessä ohjelmistokehittäjä näkee selvästi työnsä tuloksen. Jos testit eivät mene läpi, se on merkki siitä, että ohjelmakoodia pitää vielä parantaa. Kehittäjä voi luottaa siihen, että toteutettu osio vastaa ohjelmistomäärittelyjä, kun testit suoritetaan onnistuneesti. TDD ohjaa kehittäjää etenemään pienin askelin. On huomattavasti tuottavampaa paloitella toteutettavat osiot pieniin yksiköihin kuin yrittää ohjelmoida iso palanen kerrallaan. Kun kehitystyössä huomaa, että jokin kohta aiheuttaa virheen, ongelman etsiminen on helpompaa, kun uutta ohjelmakoodia on syntynyt vain pienen toiminnallisuuden osilta.

2.5.1 Testivetoinen ohjelmistokehityssykli

Ohjelmistokehittäjät hyödyntävät TDD menetelmää kukin omalla tavallaan. Pääpiirteissään prosessi noudattaa viiden vaiheen ohjelmistokehityssykliä, jonka vaiheet Horch kuvaa kirjassaan (Horch, 2004):

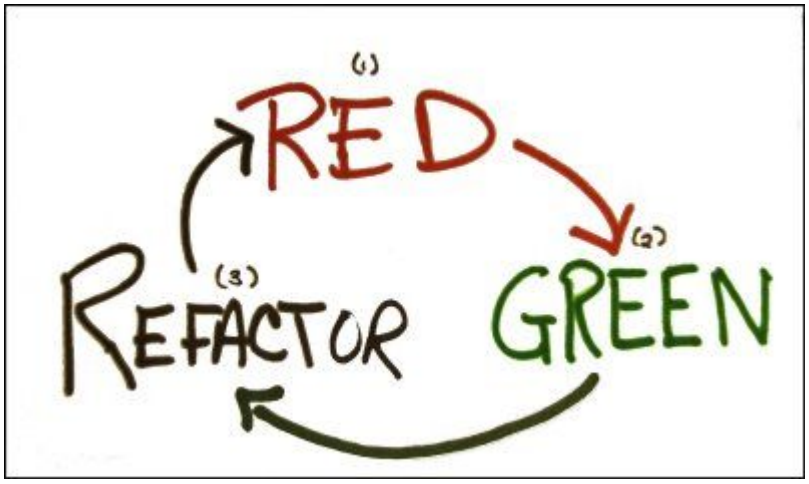
1. Kirjoita yksikkötesti uudelle, toteuttamattomalle toiminnolle.

2. Suorita aiemmin lisätyt testit sekä kohdassa 1 lisätyt uudet testit ja varmista, että uudet testit eivät mene läpi.
3. Kirjoita uutta toimintoa vastaava ohjelmakoodi.
4. Aja kaikki yksikkötestit uudelleen ja tarkista, että myös uudet testit menevät läpi.
5. Refaktoroi ohjelmakoodia.

Seuraavissa kappaleissa kuvataan listan vaiheita tarkemmin pohjautuen Horchin kirjaan sekä Shoren blogitekstiin (Horch, 2004; Shore, 2005).

Testivetoisessa ohjelmistokehityssyklissä uuden ominaisuuden toteuttaminen alkaa aina testin kirjoittamisella. Jotta testin voi suunnitella, on ymmärrettävä tarkasti toiminnon määrittely ja vaatimukset. Tämä on usein testivetoisen ohjelmistokehityssyklin vaativin osa. Tarkoituksena on pystyä valitsemaan ja toteuttamaan sellainen testi, joka ohjaa kohti toivottua lopputulosta. Tämä ensimmäinen vaihe ohjaa kehittäjää ajattelemaan ohjelmistoa enemmän käyttäjän ja rajapintojen näkökulmasta, ei niinkään ohjelman sisäisen rakenteen kautta.

Kun yksikkötestit on kirjoitettu, suoritetaan testit. Toivottu lopputulos on, että nämä uudet testit eivät mene läpi. Testiajolla todennetaan, ettei kyseistä toiminnallisuutta löydy jo toteutettuna ohjelmistosta. Testin läpimeno voi johtua myös testin virheellisyydestä, jonka vuoksi testi menee aina läpi. Ohjelmistokehityssyklin tästä vaiheesta voidaan käyttää termiä *Red* (ks. Kuva 2). Väri symboloi testien läpimenoa ja samaa väriä käytetään myös käytännön työssä testitulosten yhteydessä. Vihreä merkitsee testien onnistumista, punainen väri kiinnittää huomion epäonnistuneeseen testiin.



Kuva 2. *TDD-iteraatiosykli (Martin, 2014)*

Toteutusvaiheessa toteutetaan vain testitapauksen kuvaama osa, kiinnittämättä sen suurempaa huomiota ohjelmistokoodin tyyliin tai rakenteeseen. Vaiheen ainoana päämääränä on saada aikaiseksi koodi, joka läpäisee testit. Ohjelmistokehittäjän tulee kiinnittää huomiota siihen, että ohjelmakoodi ei toteuta mitään muuta kuin ne toiminnallisuudet,

joita testitapaukset testaavat. Tässä vaiheessa hyväksytään, että koodin luettavuus ja kehittämistyyli eivät välttämättä ole lopullisessa muodossaan, niihin palataan kehityssyklin myöhemmässä vaiheessa.

Kun ohjelmakoodi on kirjoitettu, siirrytään vaiheeseen 4 ja ajetaan testit uudelleen. Jos kaikki testitapaukset menevät läpi, voidaan olettaa, että uusi toiminnallisuus on tehty vaatimusmäärittelyn mukaisesti ja ollaan Kuvan 2 esittämässä vaiheessa *Green*. Mikäli testit eivät vielä mene läpi, muutetaan ohjelmakoodia ja toistetaan testit. Näitä vaiheita toistetaan niin kauan, että testit onnistuvat.

Refaktoroinnin jälkeen ajetaan samat testit uudelleen ja todennetaan, että ohjelmakoodin muokkauksen jälkeen testit menevät edelleen läpi. Koska testit suoritetaan aina koodimuokkauksen jälkeen, kehittäjä uskaltaa tehdä rakenteeseen isojakin muutoksia. Testit ilmentävät heti, mikäli muutokset aiheuttivat ongelmia toiminnallisuuteen ja testien avulla myös ongelman paikantaminen helpottuu.

Toiminnallisuuden toteutuksessa ohjelmistokehittäjä toistaa Kuvassa 2 esitettyä iteraatiosykliä niin kauan, että toiminnallisuus on valmis ja koodin rakenne on tarpeeksi selkeä. Viisivaiheinen ohjelmistokehityssykli aloitetaan alusta, kun aletaan toteuttamaan seuraavaa, uutta ohjelman toiminnallisuutta.

2.5.2 TDD-kehitysmenetelmän edut

Testivetoisen ohjelmistokehityksen hyödyiksi Acharya listaa kirjassaan seuraavat pääkohdat (Acharya, 2013, s.30):

1. Menetelmällä tuotetaan puhdasta, testikelpoista ja ylläpidettävää ohjelmakoodia.
2. Kun ohjelmistoa kehitetään inkrementaalisesti, kehitystyö on helpompaa, koska koodia kirjoitetaan ja käytetään samanaikaisesti
3. Kun ohjelmakoodi dokumentoidaan toteutuksen yhteydessä, dokumentoinnin päivittäminen usein unohtuu, kun myöhemmin tulee tarve päivittää koodia. Dokumentoinnin ja sen päivittämisen vaihtoehtona on, että koodi ja yksikkötestit voidaan kirjoittaa niin selkeiksi, että kuka tahansa koodinlukutaitoinen henkilö ymmärtää koodin tarkoituksen näitä silmäilemällä. TDD-menetelmällä toteutetut testit kirjoitetaan siten, että ne tarjoavat tarpeeksi ohjelmakoodin dokumentaatiota. Koska testi on samalla dokumentaatio, testejä täytyy ylläpitää ja siistiä, jotta ne pysyvät luettavina ja jatkokehityskelpoisina.
4. Koodin oikeellisuuden todentamiseksi kannattaa kirjoittaa useita testejä liittyen nollan, negatiivisten lukujen ja muiden raja-arvojen testaukseen. Testeillä yritetään murtaa oma ohjelmakoodi mahdollisimman varhaisessa vaiheessa. Ohjelmapakettia ei tarvitse toimittaa laadun tarkastukseen tai asiakkaalle asti, jotta koodivirheet tulevat ilmi.

5. On tärkeää välttää liiallista ohjelmakoodin kirjoittamista. Riittää, että koodia on kirjoitettu vain sen verran, että testin suorittaminen näyttää vihreää.

Acharyan mukaan edellä esitetyt edut saavutetaan käyttämällä TDD-menetelmää sen sijaan, että yksikkötestit kirjoitettaisiin jälkikäteen ohjelmakoodin valmistuttua. Listauksessa esitetyt edut nousevat esille myös monissa muissa TDD-menetelmää tutkivissa ja esittelevissä kirjoissa ja artikkeleissa. TDD-menetelmän etuja verrattuna TLD-kehitystapaan on kuitenkin hankala todentaa, toteaa Vestola kandidaatintyössään (Vestola, 2008). Vestola on tutkinut taustamateriaaleja ja löytänyt vertailevia empiirisiä tutkimuksia suhteellisen vähän. Akateemisissa tutkimuksissa voidaan järjestää testitilanteita, jossa eri ohjelmoitsijat ohjelmoivat saman toiminnallisuuden TDD- tai TLD-menetelmällä ja nämä tuotokset ovat keskenään pitkälti vertailukelpoiset. Työelämässä tuotanto toimii kuitenkin siten, että lopputuote toteutetaan joko oikealle asiakkaalle tilattuna työnä tai vastaavasti yrityksen omana tuotekehitysprojektina. Yrityksillä ei useinkaan ole taloudellisesti mahdollista tai kiinnostusta toteuttaa ohjelmointia tutkimuksellisista syistä eri kehitystapojen etujen selvittämiseksi, koska työmäärä on kaksinkertainen. Pitää myös muistaa, että kahden eri kehitystavan vertailtavuus on haastavaa johtuen eri ohjelmoitsijoiden tasoeroista.

Kun noudattaa TDD-menetelmän kehityssykliä, jokaiseen ohjelmiston toiminnallisuuteen syntyy testitapauksia. Martinin mukaan (Martin, 2007) yhden kehittäjän työpanoksena testitapauksia kertyy kymmeniä päivässä ja useita tuhansia vuodessa ja testit kattavat yli 90 prosenttia tuotantokoodista. Jos testien kattavuus saadaan näin korkeaksi, Martinin kokemuksen mukaan suuretkin ohjelmakoodin siistimiset ja muutokset voidaan tehdä ilman suurta riskiä ohjelman rikkoutumisesta. Testien ajaminen muutosten välissä ja jälkeen kertoo välittömästi, jos joku toiminnallisuus meni epäkuntoon.

3. PROFIT PLP JA COM -OHJELMISTOT JA OHJELMOINTIRAJAPINNAT

Ohjelmointirajapinta (Application Programming Interface, API) voidaan kuvailla siltana, jonka yli ohjelmistot keskustelevat keskenään ja vaihtavat tietoja. Kun puhutaan rajapinnasta, liittyy se itsenäisiin ohjelmistoihin, jotka jakavat ohjelman sisäisiä tietoja keskenään rajapintamäärittelyn toteuttavalla tekniikalla. Ohjelmistot voivat olla toteutettu keskenään eri ohjelmointitekniikoilla ja tietokantaratkaisuilla. Tärkeää on vain se, että keskusteluyhteys eli rajapinta on toteutettu ennalta määritellyllä tavalla, yleensä standardimuotoisena tiedostomuotona. Ohjelmistointegraatiosta voidaan puhua tilanteessa, jossa ohjelmistot toimivat kiinteämmin yhteen. Ne voivat jakaa saman tietokannan kokonaan tai osittain ja käyttää samaa koodipohjaa. Integroiduissa järjestelmissä tietojen jakaminen on reaaliaikaista eikä järjestelmien välillä tarvita tietojen synkronointia tai eräajopohjaisia tietojen vienti- ja tuontitoiminnallisuuksia.

Ohjelmistorajapintojen toteutuksessa on tyypillistä tietojen vaihto eräajopohjaisesti. Toiseen ohjelmistoon vietävät tiedot voidaan muodostaa tarvittaessa, tai tiedot voivat siirtyä aikataulutetusti esimerkiksi päivittäin, viikoittain tai kuukausittain. Tiedonvaihdon tiheys on aina tapauskohtaista ja se perustuu tiedonvälittämisen vaatimuksiin eli siihen, kuinka nopeasti järjestelmässä syntynyt tieto pitää olla toisen järjestelmän saatavilla. Järjestelmässä voidaan muodostaa esimerkiksi veroraportti taloushallinnon tarpeisiin kerran vuodessa. Jos kyse on esimerkiksi asiakastietojärjestelmästä ja näitä asiakastietoja hyödyntävästä erillisestä laskutusjärjestelmästä, on selvää, että tietojen synkronointi järjestelmien välillä on hyvä tehdä tunneittain tai päivittäin.

Ohjelmistorajapinnat voidaan toteuttaa avoimina tai rajoitettuina. Avoimia rajapintoja käytetään tyypillisesti esimerkiksi valtion ja kuntien tuottamien tietojen jakamiseen. Avointa dataa on nykyisin runsaasti saatavilla liittyen esimerkiksi paikkatietoihin, sosiaali- ja terveysalan tuottamaan tietoon sekä kuntien ja valtion keräämään tietoon. Ohjelmistotoimittajat voivat hyödyntää avoimen rajapinnan dataa omissa järjestelmäratkaisuissaan. Yrityksen sisällä tuotettuun tai henkilötietoihin liittyvään dataan on yleensä rajoitetumpi pääsy. Tietojen lukemiseen tarvitaan käyttöoikeudet ja käyttäjän autentikoinnilla varmistetaan pääsy vain niihin tietoihin, joihin käyttäjä on oikeutettu. Yhä useammin ohjelmistotoimittajilta vaaditaan kuitenkin rajapintatuki toimittamiinsa ohjelmistoihin, jotta järjestelmässä tuotettua tietoa voidaan hyödyntää organisaation muissa järjestelmissä. Aikaisemmin järjestelmät olivat pääasiassa suljettuja ja samoja tietoja jouduttiin ylläpitämään useassa eri järjestelmässä. Tällöin tietojen siirto oli lähinnä manuaalisten toimenpiteiden varassa.

3.1 Profit Life & Pension, PLP

Työn tilaaja, Profit Software Oy, on vuonna 1992 perustettu, ohjelmistoratkaisuja ja konsulttipalveluita finanssialalle tarjoava suomalainen yritys. Yrityksen asiakkaina on tämän päättötyön kirjoitusvuonna 2018 muun muassa Osuuspankki, Mandatum Life, Handelsbanken ja Säästöpankki Suomen markkinoilla sekä Ruotsin markkinoilla Länsförsäkringar. Tämän päättötyön yhteydessä on tarkoitus kehittää yrityksen kahden oman ohjelmistotuotteen välille rajapinta, jonka avulla vakuutussopimusten korvausmaksumääräykset siirtyvät ohjelmistojen välillä.

Päättötyön tilaajan lippulaivaohjelmisto Profit Life & Pension (PLP) kattaa vakuutussopimusten koko elinkaarihallinnan. PLP-ohjelmiston vakuutussopimustyyppejä ovat säästö- ja eläkevakuutukset, joissa päämotiivina on säästäminen. PLP kattaa kaikki vakuutushallinnan operatiiviset perustoiminnot tarjousta edeltävistä alustavista laskelmista aina sopimuksen päättäviin korvaustoimintoihin saakka.

Pääpiirteissään vakuutussäästäminen etenee tarjouksen tekemisen ja hyväksymisen kautta sopimuksen syntymiseen. Asiakas maksaa sopimuksen mukaisia maksuja säästöaikana joko kertaluonteisesti tai toistuvasti esimerkiksi kuukausittain. Pääoma on sidoksissa säästöaikana hajautetusti erilaisiin rahastoihin. Kun säästöaika päättyy, siirrytään korvaustoiminnallisuuteen. Korvauksen maksu voidaan maksaa yhdessä erässä tai se voi ajoittua useammalle vuodelle toistuvaismaksuina. Sopimus päättyy, kun säästöpääoma on maksettu ja sopimuksella ei ole enää rahaa kirjattuna.

Sopimushallinnassa ja korvaustoiminnallisuuden keskiössä on säästön ja sijoituskohteiden hallinta yhtiötä suojaavalla tavalla. Kun korvausmäärä irrotetaan sopimukseen kertyneestä säästöstä, joka puolestaan on sidoksissa rahastoihin tai vastaaviin sijoitusinstrumentteihin, korvaus realisoidaan käyttämällä kurssiviiveitä. Tämä tarkoittaa käytännössä vahvistettujen kurssien käyttämistä eli esimerkiksi viiden arkipäivän takaista kurssitilannetta, jolloin kurssiheilahtelut eivät muodosta riskiä sopimuksen myöntäneelle yritykselle.

PLP on kehitetty alun perin pilottiasiakkaiden ja potentiaalisten asiakkaiden toiveiden pohjalta. Heidän järjestelmissään korvausprosessi ja sopimusten hallinta olivat tiukasti erillään toisistaan, mikä sopi melko hyvin korkoperusteisiin vakuutustuotteisiin. Kun Suomessa alkoi sijoitussidonnaisten säästö- ja eläkevakuutusten myynti, haluttiin kattaa koko vakuutussopimusten elinkaari samassa järjestelmässä. Korvauksia varten tarvittava ulkoisiin sijoituskohteisiin sidotun säästön realisointi oli joka tapauksessa sopimusjärjestelmän vastuulla, joten korvausmaksatus oli luontevaa yhdistää samaan sopimuksenhallintajärjestelmään. Korvaustilanteeseen liittyvät asiakastulosteet ja raportointi voidaan PLP-ohjelmistossa tuottaa yhdessä paikassa ja haluttaessa hyvin tarkalla tietosisällöllä, kun koko sopimukseen liittyvä data on saatavilla.

3.2 Claim Payout Manager, COM

Korvausmaksatukseen kehitetty Claim Payout Manager (COM) toteuttaa korvausmaksatuspuolen jättäen sopimusten hallinnan muiden järjestelmien hoidettavaksi. COM on kehitetty ensisijaisesti Ruotsin markkinoille, jossa työeläkejärjestelmä eroaa jonkin verran Suomen mallista. Suomessa eläkkeen maksatus hoidetaan ’viimeisen luukun’ periaatteella, jossa työntekijän viimeisen työsuhteen hoitanut eläkeyhtiö hoitaa koko työhistorian perusteella kertyneen eläkkeen hallinnan ja maksatuksen. Ruotsissa hajautetumman työeläkejärjestelmän takia on tavallista, että asiakkaalle maksetaan työeläkettä työhistoriassa monen eri työnantajan kautta kertyneiden työeläkesopimusten perusteella. Koska päättötyön tilaajan ensimmäinen ruotsalainen asiakas oli ennestään tottunut jakamaan korvauskäsittelyn vastuut järjestelmäkokonaisuudessaan sopimuksenhallintajärjestelmiin ja erilliseen korvausjärjestelmään, oli luontevaa kehittää keskitetty korvausjärjestelmä palvelemaan tätä tarvetta.

Korvausten ulosmaksatusjärjestelmä COM tietää vain minimaalisesti vakuutussopimuksista, joiden perusteella maksatus tapahtuu. Käytännössä sopimuksista toimitetaan COM-ohjelmiston tarjoaman sisäänlukurajapinnan kautta vain suppeat korvauksen avaustiedot ja bruttokorvausmäärä. Korvausprosessin muu osa hoidetaan COM-ohjelmiston puolella. Siellä hallitaan esimerkiksi edun- ja maksunsaajien tiedot, verotus- ja verokorttikäsittelyt, korvaussumman jaot useamman edunsaajan kesken sekä tarvittavat raportoinnit. Keskitetyn ulosmaksatuksen etu on, että maksuja voidaan niputtaa yli sopimusrajojen. Eläkkeensaaja voi näin ollen saada kuukausittain yhden maksusuorituksen, johon voi olla kerätty maksatuksia useista eri sopimuksista. COM-ohjelmistossa on korvausmaksatukseen liittyen jonkin verran lisätoimintoja Suomen markkinoilla olevaan PLP-ohjelmistoon verrattuna sekä siihen on toteutettu Ruotsissa vaadittavat viranomais- ja maksuliikenneliittymät sekä verotussäännöt.

3.3 COM ja PLP -ohjelmistojen kehitystarpeet

Päättötyön tilaaja haluaa tarjota sopimuksenhallintaohjelmistoa Ruotsin markkinoille siellä käytössä olevan COM-ohjelmiston rinnalle. Myös Suomen markkinoilla uusilla asiakkailla on ollut kiinnostusta jakaa sopimus- ja korvausprosessivastuut eri sovelluksiin. Koska Ruotsin markkinoilla on ennestään totuttu toimintamalliin, jossa sopimustenhallinta ja korvauskäsittely hoidetaan kumpikin omissa järjestelmissään, on luontevaa lähteä kehittämään PLP-ohjelmistosta versiota, jossa korvauskäsittelytoiminnot voidaan halutaessa hoitaa COM-ohjelmistossa. Ruotsin vakuutusalan toimintoihin liittyy lisäksi maakohtaisia verotuksellisia erityispiirteitä ja muita viranomaiskäytäntöjä, jotka löytyvät COM-ohjelmistosta valmiina. Suomen markkinoilla jatketaan PLP-ohjelmiston käyttöä myös nykymuotoisena versiona, jossa korvaustoiminnot on integroitu sopimusprosessin kanssa samaan ohjelmistoon, minkä vuoksi korvauskäsittelyn ohittaminen halutaan tarjota PLP-ohjelmistoon valinnaisena toiminnallisuutena.

Tämän päättötyön yhteydessä PLP-ohjelmistoon toteutetaan toiminnallisuus, jossa vakuutustuotekohtaisesti voidaan parametriohjatusti sivuuttaa PLP-ohjelmiston sopimushallinnassa korvaustoiminnallisuus. Kun tällaisen vakuutustuotteen sisältävä sopimus siirtyy korvaustilaan, korvaus ja siihen liittyvät maksuerät siirretään COM-ohjelmiston rajapintamäärittelyn mukaisina siirtotiedostoina COM-ohjelmiston korvauskäsittelyprosessiin. Koska COM-ohjelmistoon tuodaan jo ennestään korvausmääräyksiä eri sopimuksenhallintaohjelmistoista, COM tarjoaa valmiin rajapinnan korvausmääräysten tuontiin. Toiminnallisuuden toteutus ja testaus on kuvattu tarkemmin luvussa 4 *Rajapinnan toteutus, testaus ja testitulokset*.

Työssä on tarkoitus säilyttää PLP ja COM -ohjelmistot erillisinä sovelluksina ja tehdä korvaustapahtumien siirrot rajapinnan kautta. Ohjelmistot säilyvät itsenäisinä, jolloin kumpaakin voidaan käyttää yksinään ilman toisen ohjelmiston tukea. Tietojen siirto järjestelmien välillä on yksisuuntaista PLP-ohjelmistosta COM-ohjelmistoon ja siirtotiedostojen muodostaminen toteutetaan eräajopohjaisesti. Tällaisesta rajapinnasta käytetään nimitystä löyhä sidonta (*loose coupling*), jossa riippuvuudet ohjelmistojen välillä ovat minimissään. Kiinteämpi integrointi voi olla mahdollista järjestelmäkehityksen myöhemmässä vaiheessa. Toiminnallisuuden suunnittelussa ja toteutuksessa käytetään Open/Closed metodiikkaa (ks. 2.4 Ohjelmiston lähdekoodin refaktorointi), jossa järjestelmä jätetään avoimeksi laajennuksille.

3.4 COM-ohjelmistossa käytetyt rajapintatekniikat

Ohjelmistorajapintojen toteutukseen on vakiintunut eri tekniikoihin perustuvia rajapintamäärityksiä, joista yleisesti käytetyimpiä tekniikoita ovat eXtensible Markup Language (XML) ja JavaScript Object Notation (JSON) (Vázquez, 2015). COM-ohjelmisto sisältää valmiit rajapinnat sopimuksen perustietojen sekä korvausmääräysten tuonnille. Ohjelmiston siirtorajapinta on toteutettu XML-tekniikalla, joka on tekstiperusteinen rakenteellinen merkintäkieli (*markup language*) (Patni, 2017). XML-kieli on yksi vakiintuneista ja laajasti käytetyistä tiedonsiirtoformaateista ja se on suunniteltu datan säilyttämiseen ja tiedon välitykseen. XML-muotoinen tiedosto voidaan muodostaa toisen järjestelmän pyynnöstä tai se voidaan muodostaa valmiiksi tiettynä ajanhetkenä eräajopohjaisesti.

3.4.1 XML-rajapinta

XML-kieli muistuttaa Hypertext Markup Language (HTML) -kuvauskieltä, joka on suunniteltu tiedon näyttämiseen. HTML-kielessä tagien nimeäminen ja käyttötarkoitus on ennalta sovittu. Esimerkiksi <table> merkitsee taulukkoa, <p> kappaletta ja <h1> on sivun pääotsikko. Tiedon lukijalle, joka on yleensä www-selain, pitää kertoa päättävällä tagilla, missä kyseinen tieto päättyy.

Tarkastellaan esimerkkinä HTML-tagien käyttöä. Esimerkissä sivulle annetaan pääotsikko ja ensimmäinen kappale:

```
<h1>Sivun pääotsikko</h1>
<p>Kappaleen tekstiä, teksti jatkuu...</p>
```

HTML-kuvauskielessä dokumentti voidaan avata selaimella ja useat selainohjelmat osaavat näyttää tiedoston sisällön, vaikka kaikkia päättäviä tageja ei löytyisikään, tosin siinä tapauksessa lopputulos ei välttämättä näytä halutulta.

XML-kielessä voidaan käyttää tagien nimeämistä vapaammin. Kun rajapinta toteutetaan XML-tekniikalla, rajapinnan laatija päättää tagien nimeämisestä käyttäen tagin tietosisältöä kuvaavaa nimeämistapaa. XML-rajapinnan käyttäjien pitää tietää kyseiseen rajapintaan liittyvät nimeämiskäytännöt voidakseen käyttää rajapintaa. Suurin ero HTML ja XML -tiedostojen välillä on, että XML-dokumentti pitää olla muodostettu oikein säännönmukaisesti (Patni, 2017). Solmun avaavalle tagille pitää löytyä päättävä tagi ja tagien pitää noudattaa hierarkiaa. Ohjelmassa 3 esitellään COM-ohjelmistoon siirtyvää XML-muotoista korvausmaksumääräystä.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns8:paymentMessagesFileType>
    <ns8:paymentFileHeader>
        <ns7:fileId>payment-orders-2018-04-08T09:14:46.713</ns7:fileId>
        <ns7:systemId>PLP</ns7:systemId>
        <ns7:creationTime>2018-04-08T09:14:46.713+03:00</ns7:creationTime>
    </ns8:paymentFileHeader>
    <ns8:paymentMessage>
        <ns6:policyId>2500000016</ns6:policyId>
        <ns6:claimType>maturity</ns6:claimType>
        <ns6:paymentOrder>
            <ns6:dueDate>2018-04-10</ns6:dueDate>
            <ns6:grossAmount>8593.3</ns6:grossAmount>
            <ns6:taxCode>K</ns6:taxCode>
            <ns6:firstPayment>true</ns6:firstPayment>
        </ns6:paymentOrder>
    </ns8:paymentMessage>
    <ns8:paymentFileFooter>
        <ns7:fileId>payment-orders-2018-04-08T09:14:46.713</ns7:fileId>
        <ns7:systemId>PLP</ns7:systemId>
        <ns7:messageCount>1</ns7:messageCount>
    </ns8:paymentFileFooter>
</ns8:paymentMessagesFileType>
```

Ohjelma 3. *Esimerkki COM-ohjelmistoon siirtyvästä XML-muotoisesta korvauksen maksumääräyksestä*

Ohjelmassa 3 esitelty XML-tiedosto on rakennettu sääntöjen mukaisesti. Jokainen tietue sisältää aloittavan ja päättävän tagin ja tagit noudattavat hierarkiaa. Mikäli hierarkiassa esiintyy ristiriitoja, XML-rajapintaa käyttävä sovellus ei kykene lukemaan tietueiden sisältöjä.

Esimerkiksi merkintätapa

```
<ns8:paymentFileHeader>
        <ns7:systemId>PLP</ns8:paymentFileHeader>
</ns7:systemId>
```

aiheuttaisi ongelmia hierarkiaristiriidan vuoksi.

Tarkastellaan Ohjelman 3 tietosisältöjä tarkemmin. Dokumentti sisältää koko siirtotiedostolle yhteisiä määrittelyitä, kuten paymentFileHeader ja paymentFileFooter -tageissa määriteltävät tiedoston yleiset tunnisteet. Tiedosto sisältää myös itse korvausmaksatukseen liittyvät yksityiskohtaiset, paymentMessage-tagin sisälle määritetyt tiedot. Kun siirtotiedosto muodostetaan, mukaan poimitaan kaikki korvausmaksujonossa olevat maksut, joten todellisessa tilanteessa Ohjelman 3 paymentMessage-tagi toistuu siirtotiedostossa niin monta kertaa kuin maksumääräyksiä on siirrettävänä. Jokaisen maksumääräyksen paymentMessage-tagin sisällössä määritellään korvausmaksuun liittyvät yksityiskohdat, jolloin paymentMessage-tagin sisällä on sille alisteisia uusia tageja ja niihin liittyviä tietosisältöjä. Esimerkiksi yhteen maksumääräykseen liittyvä eräpäivä löytyy tagista dueDate,

```
<ns6:dueDate>2018-04-10</ns6:dueDate>,
```

jossa eräpäivän tietosisällöksi on määritetty 10.4.2018. Alatunnisteessa (paymentFileFooter) kerrotaan vielä vastaanottavalle ohjelmistolle messageCount-tagilla, montako maksumääräystä kyseisessä tiedostossa on mukana. Vastaanottava COM-ohjelmisto tarkistaa siirron yhteydessä, että toimitettujen maksumääräysten lukumäärä täsmää messageCount-tagin antamaan lukumäärään.

XML-tiedosto alkaa aina määrittelyllä eli prologilla

```
<?xml version="1.0"?>,
```

jossa versiotieto on pakollinen. Tiedoston käyttäjän pitää tietää, mitä XML-versiota tiedostossa on käytetty. Lisämääreinä määrittelylle voidaan antaa merkkikielen määrittelevä tieto, `encoding`, joka XML-tiedostoissa on oletuksena `compressed Unicode` (UTF-8), sekä riippuvuustieto `standalone`. Riippuvuustiedoksi annetaan ”yes”, mikäli tiedosto toimii itsenäisesti eikä viittaa ulkopuoliseen kokonaisuuteen.

3.4.2 JAXB-ohjelmointirajapinta

Korvausmaksatuksen siirtotoiminnallisuudessa Ohjelmassa 3 esitelty XML-muotoinen siirtotiedosto on lopputulos, joka pitää ensin muodostaa ohjelmallisesti. Java-kielessä on olemassa työtä helpottava Java Architecture for XML Binding (JAXB) -ohjelmointirajapinta, jonka avulla XML-tiedostoja luodaan, luetaan ja kirjoitetaan. Rajapinnassa käytettävien XML-tiedostojen rakenne esitellään mallissa (schema), jossa määritellään tiedos-

tossa sallitut komponentit ja attribuutit, niiden suhteet, järjestys ja komponenttien tietotyypit (Fallside *et al.*, 2018). Jotta JAXB-ohjelmointirajapintaa voidaan hyödyntää, JAXB tarvitsee toimiakseen mallin, joka on toteutettu Fallside *et al.* artikkelissa esiteltyjen W3C-suositusten mukaisena XML Schema Language -mallina.

Ohjelmassa 4 esitellään korvausmaksatukseen liittyvän siirtorajapinnan määrittelyyn toteutettu malli, jota hyödynnetään JAXB-ohjelmointirajapinnan kautta. Ohjelmassa 3 on esitelty muodostettava XML-dokumentti laajemmin kuin Ohjelman 4 mallissa on esitelty, jotta esimerkki pysyy suppeampana ja helpommin tulkittavana. Siksi Ohjelmassa 4 määritellään XML-dokumentin rakenne vain korvausmaksatukseen liittyvän paymentOrder-komponentin ja sen alikomponenttien osalta.

```
<xsd:complexType name="paymentOrderType">
  <xsd:annotation>
     <xsd:documentation>
        Type for data needed when adding a claim payment to a claim.
     </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
     <xsd:element name="dueDate" type="xsd:date"/>
     <xsd:element name="grossAmount" type="common:moneyType"/>
     <xsd:element name="taxCode" type="common:taxCode"/>
     <xsd:element name="firstPayment" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
```

Ohjelma 4. *XML-malli, jossa määritellään korvausmaksatuksen siirtotiedoston muodostamisessa tarvittavia tietoja*

Komponentin paymentOrder alikomponentteina näkyvät korvausmaksun eräpäivä (dueDate), summa (grossAmount), verokoodi (taxCode) ja onko kyseessä ensimmäinen sopimukseen liittyvä korvauksen maksu (firstPayment). Alikomponenteille on määritetty tietotyypit (type), kuten päivä (date) tai totuusarvo (boolean). Tietotyyppejä voidaan määritellä myös perustietotyyppejä monipuolisemmin erityistarpeisiin sopiviksi, kuten verokoodin osalta on tehty. Tyyppi common:taxCode viittaa toiseen määrittelytiedostoon, jossa verokoodin sallitut arvot on määritelty tarkemmalla tasolla. Esimerkiksi tässä tapauksessa määrittelytiedostossa on rajattu verokoodin sallituiksi arvoiksi muutama kirjainlyhenne, joka viittaa käytössä oleviin verokantoihin. Verokoodin tietosisällöksi ei voi asettaa muita kirjainlyhenteitä kyseisen määrittelyn ulkopuolelta.

Mallin sitominen (binding) tarkoittaa Java-luokkien muodostamista mallin määrittelyiden mukaisesti. JAXB-toteutukset sisältävät mallin sitomisen toteuttavan tekniikan (binding compiler) automaattiseen Java-luokkien muodostamiseen kuten mallissa on määritelty (Mehta *et al.*, 2003). Mallin sitomisessa muodostetut Java-luokat käyttäytyvät samoin kuin muilla menetelmillä muodostetut luokat. Niiden avulla luodaan olioita, talletetaan tietoja olion attribuutteihin, muutetaan olion tilaa sekä käytetään olion metodeja ohjelmiston suorituksen yhteydessä.

Tietojen luku XML-tiedostoista Javan olioiksi toteutetaan myös JAXB-tekniikalla. Tietojen lukeminen tehdään JAXB-tekniikan toteuttaman un-marshalling-toiminnon avulla mallin sitomisen yhteydessä muodostettuihin Java-luokkien toteuttamiin olioihin. Tämän jälkeen oliot käyttäytyvät kuten olioiden kuuluukin ja olioiden tilaa voidaan muuttaa ohjelman suorituksen aikana. Kun olioihin on tehty halutut muutokset ohjelmistossa, oliot voidaan tallentaa takaisin xml-tiedostoiksi JAXB-tekniikan marshalling-toiminnolla. Unmarshalling ja marshalling -tekniikoita voi käyttää tarpeen mukaan itsenäisesti, toisistaan riippumattomasti. Toisen tekniikan voi jättää hyödyntämättä, mikäli sen tarjoamia palveluita ei tarvita. Kun korvausmaksumääräyksiä tuodaan PLP-ohjelmistosta COM-ohjelmistoon, tuonnissa ei tarvita un-marshalling-tekniikkaa, koska tiedot olioihin luetaan PLP-ohjelmiston tietokannasta. Tietojen tallennuksessa XML-tiedostoksi sen sijaan hyödynnetään marshalling-tekniikkaa.

4. RAJAPINNAN TOTEUTUS, TESTAUS JA TESTITULOKSET

Toteutettava toiminnallisuus vaatii muutoksia PLP-ohjelmistoon. Kun määriteltyjä vakuutustuotteita käyttävät vakuutussopimukset siirtyvät korvaustilaan, niiden tulee ohittaa PLP-ohjelmiston omat korvaustoiminnallisuudet. Näiden korvausten ulosmaksu siirtyy COM-ohjelmiston korvauskäsittelyyn. COM-ohjelmiston rajapintaa ei tässä yhteydessä ole tarve muuttaa, koska se sisältää jo ennestään riittävät siirtorajapinnalta vaaditut toiminnot. Toiminnallisuuden toteutukseen osallistunut toteutustiimi, tämän päättötyön tekijä mukaan lukien, osallistui toiminnon määrittelytyöhön ja kukin tiimin jäsen vastasi oman ohjelmisto-osionsa toteutustyöstä. Muutoksia tuli PLP-ohjelmiston osioihin, joissa vakuutussopimuksia perustetaan, käsitellään ja siirretään korvaustilaan. Sopimuksiin liittyviin eräajopohjaisesti toteutettuihin tilanmuutoksiin, ylläpitotoimintoihin ja siirtotiedoston muodostukseen tarvittiin myös muutoksia. Uusien vakuutustuotteiden perustaminen ja osa toimintojen ohjauksesta tehtiin nykyisen parametrirakenteen mukaisena, ainoastaan lisäämällä tarvittavat tiedot parametristoon. Jotta COM-ohjelmisto voi vastaanottaa PLP-ohjelmistosta siirrettävät korvausmääräykset, COM-ohjelmistoon piti lisätä vastaavat vakuutussopimustuotteet. Tämä hoitui lisäämällä COM-ohjelmiston parametristoon tarvittavat tuotteet.

Päättötyön ohjelmointitoteutukseen kuuluu toteutettavasta toiminnallisuudesta ne osiot, jossa korvaustilassa olevien sopimusten ulosmaksu siirretään COM-ohjelmiston rajapinnan määrittelyjen mukaisiksi siirtotiedostoiksi. Tästä johtuen tarkastelemme tarkemmin ohjelmistomuutoksia siitä hetkestä eteenpäin, kun sopimukselle on kirjattu korvausmääräys ja tiedämme, että kyseessä on sellainen vakuutustuote, joka on parametriohjatusti määritelty siirtyväksi COM-ohjelmiston korvauskäsittelyyn. Päättötyöhön liittyvä ohjelmointitoteutus poimii siirtoa odottavat maksumääräykset ja muodostaa niistä COM-rajapintamäärittelyn mukaiset XML-muotoiset siirtotiedostot. Kun siirtotiedostot on muodostettu, niiden lukeminen COM-ohjelmistoon sisään on COM-ohjelmiston rutiinitoiminnallisuus. Sisäänlukutoiminnallisuudet sekä korvausprosessin eteneminen COM-ohjelmiston korvauskäsittelyssä jätetään päättötyön tarkastelun ulkopuolelle.

Työn tilaajan prosessikehitystiimin yhtenä aktiivisesti kehitettävänä osa-alueena päättötyön teon aikana oli kaikkien yhtiön käytössä olevien testimenetelmien arviointi ja kehittäminen. Yksikkötestien kehittämisessä tarkasteltiin testien kattavuutta ja määriteltiin liiketoiminnan kannalta ohjelmiston tärkeimmät osat, joissa yksikkötestien kattavuus pitää olla korkein. Samalla haluttiin tutkia tarkemmin testivetoisen ohjelmistokehityksen soveltuvuutta PLP-ohjelmiston järjestelmäkehitystyöskentelyyn. Päättötyöhön liittyvä ohjelmointityö toteutettiin tästä syystä TDD-menetelmällä ja työn tarkastelunäkökulmaksi valikoitui työskentelymenetelmän arviointi ja ohjelmiston sisäisen laadun tarkastelu.

4.1 Ohjelmistokehitys- ja yksikkötestiajoympäristö

Päättötyön tilaajan ohjelmistokehittäjien yleisesti käyttämät ohjelmistokehitysympäristöt (Integrated Development Environment, IDE) Java-kielellä toteutetuissa projekteissa ovat Eclipse OXYGEN ja IntelliJ IDEA, johon myöhemmin viitataan nimellä IntelliJ. Päättötyön kehitysympäristöksi valikoitui IntelliJ, koska ympäristö oli työn toteuttajalle ennestään tuttu. IntelliJ tarjoaa yksikkötestien suorittamiseksi visuaalisesti havainnolliset työkalut, joiden avulla yksikkötestejä voidaan suorittaa sekä tarkastella testien läpimenoa.

Apache Maven on ohjelmiston kääntämiseen ja hallinnointiin kehitetty, avoimen lähdekoodin työkalu Java-ohjelmointikielellä toteutetuille projekteille (The Apache Software Foundation, 2018). Työkalua on laajennettu Maven Surefire Plugin:illa, joka suorittaa kääntämisen yhteydessä ohjelmistoon määritetyt yksikkötestit ja raportoi testitulokset.

PLP-ohjelmisto on palvelimella suoritettava sovellus, jota loppukäyttäjät käyttävät webselaimen käyttöliittymien kautta. Ohjelmistokehittäjät ajavat PLP-ohjelmistoa omilla tietokoneillaan Oracle VM VirtualBox -virtuaalikoneen avulla. Virtuaalikone emuloi todellista ajoympäristöä, tässä tapauksessa palvelinympäristöä. Ohjelmistokehittäjä lataa virtuaalikoneeseensa Docker Enterprise Edition -ohjelmistolla toteutetun palvelinympäristön, johon on asennettu PLP-ohjelmisto, DB2-tietokanta ja kaikki ohjelmiston suoritukseen tarvitut lisäosat. Kun ohjelmistokehittäjä toteuttaa PLP-ohjelmistoon uusia toimintoja, hän voi testata niitä kääntämällä Maven-työkalulla PLP-ohjelmiston lähdekoodit palvelimen ymmärtämäksi suoritettavaksi ohjelmapaketiksi. Käännön jälkeen kehittäjä siirtää muutokset sisältävän ohjelmapaketin virtuaalikoneen toteuttamaan ajoympäristöön, jonka jälkeen toimintoa voi testata selaimen kautta paikallisesti menemällä virtuaalikoneen sisäiseen ip-osoitteeseen.

4.1.1 Testien suorittaminen ohjelmistokehityksen aikana

Ohjelmistotototeutuksen yksikkötestikehyksenä käytetään TestNG-testikehystä. TestNG-testikehyksen avulla toteutetut yksikkötestit eivät tule automaattisesti yksikkötestien suoritusajoon mukaan, vaikka Java-luokan metodit olisi merkitty *@Test*-annotaatiolla ja vaikka hakemistorakenteessa noudatettaisiin suosituksen mukaista jakoa (ks. 2.2 Yksikkötestien asettaminen ja annotaatiot). Yksikkötestiluokat pitää lisätä sovellushakemiston juuressa sijaitsevaan testng.xml-nimiseen määrittelytiedostoon.

Määrittelytiedoston käyttö mahdollistaa testien ryhmittelyn ja jakamisen testikokonaisuuksiksi, mikä selkeyttää testitulosten tulkintaa. Kuvan 3 esimerkissä näytetään tarkemmin TestNG-yksikkötestikehyksen määrittelytiedoston sisältöä ja rakennetta. Kuvassa 3 esitetyn testng.xml-tiedoston rakenne mukailee PLP-ohjelmiston yhden osasovelluksen määrittelytiedoston rakennetta, mutta yksikkötestien määrää on vähennetty huomattavasti esimerkin selkeyttämiseksi.

```
<suite name="Suite1" verbose="1">
    <test name="ClaimManagerClients">
        <classes>
            <class name="com.profitsoftware.lbo.container.claims.lifeline.partialmaturity.PartialMaturity
            <class name="com.profitsoftware.lbo.savingallocation.bc.api.SavingsManagerTest" />
            <class name="com.profitsoftware.lbo.task.container.TransitionToClaimProcessorTest" />
        </classes>
    </test>

    <test name="ImportAndExportTests">
        <classes>
            <class name="com.profitsoftware.lbo.task.command.ClaimPaymentExporterTest" />
            <class name="com.profitsoftware.lbo.task.command.ClaimPaymentOrderExporterComTest"/>
            <class name="com.profitsoftware.lbo.task.command.ClaimExporterComTest"/>
            <class name="com.profitsoftware.lbo.task.command.fsbimport.FundSubImportPublisherTest" />
            <class name="com.profitsoftware.lbo.task.command.fsbimport.FundSubImporterProcessorTest" />
        </classes>
    </test>
</suite>
```

Kuva 3. *Yksikkötestien asettaminen ja ryhmittely yksikkötestikokonaisuuksiksi testng.xml-tiedoston avulla*

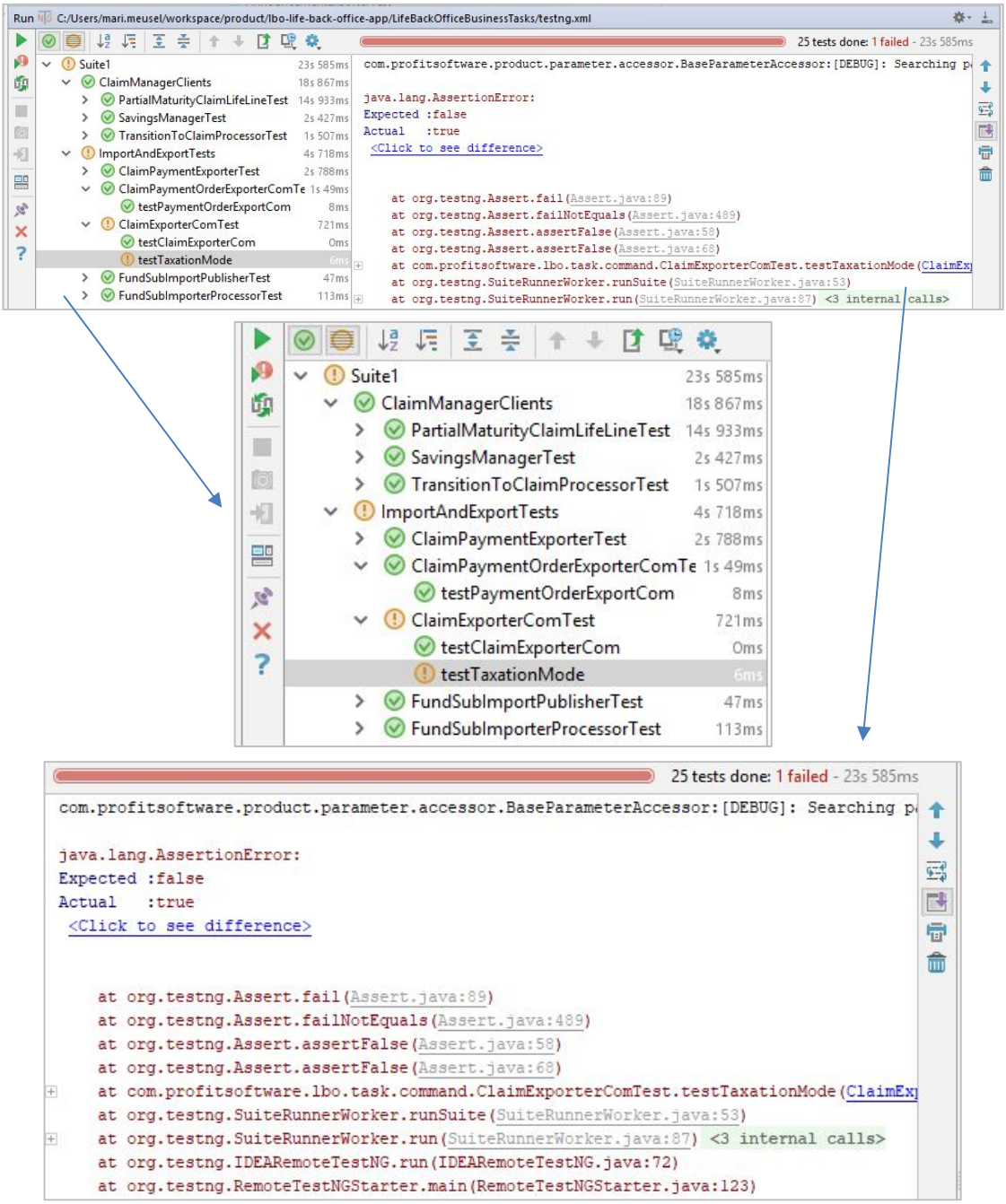
Kuvan 3 esimerkissä eräs yksikkötestikokonaisuus on nimetty ImportAndExportTests -nimiseksi. Tämän päättötyön puitteissa toteutetut testiluokat ClaimPaymentOrderExporterComTest ja ClaimExporterComTest on Kuvan 3 esimerkissä liitetty kyseiseen testikokonaisuuteen.

Yksikkötestien suorittaminen voidaan toteuttaa suoraan IntelliJ-ohjelmointiympäristön tarjoamien työkalujen avulla. Kuvan 3 määrittelytiedoston mukaisten testien suoritukseen liittyvä testitulosraportti on esitetty Kuvassa 4. Testiraportti noudattaa testng.xml-määrittelytiedoston testikokonaisuusjakoa, minkä vuoksi testin tulosta on helppo tulkita.

Kuvan 4 vasemmanpuoleisessa näkymässä näytetään suoritetut testikokonaisuudet, jotka saa laajennettua näyttämään yksittäiset yksikkötestiluokat ja niissä olevat testimetodit. Testitulosten raportoinnissa käytetään TDD-kehityssyklistä tuttuja värejä, jossa vihreällä symbolilla ilmaistaan testin onnistumista ja punaisella epäonnistumista (ks. 2.5.1 Testivetoinen ohjelmistokehityssykli). Testimetodin perässä näkyy testin suoritukseen kulunut aika ja hierarkiassa ylemmällä tasolla näkyy alempien testien yhteenlaskettu suoritusaika. Näkymästä voidaan päätellä, että ImportAndExportTests-testikokonaisuuteen kuuluvan luokan ClaimExporterComTest testimetodi testTaxationMode on epäonnistunut, kun muut testit on suoritettu onnistuneesti.

Kuvan 4 oikeanpuoleisessa näkymässä näytetään kulloinkin yhden yksikkötestin yhden metodin suoritustuloksen yksityiskohdat sen mukaan, mikä metodi on valittu vasemmasta näkymästä. Tässä tapauksessa tarkastelun kohteena on testTaxationMode-metodi ja testin tuloksen oletetaan olevan epätosi (Expected: false). Todellisuudessa testin arvo on tosi (Actual: true), jonka vuoksi testi on epäonnistunut. Ohjelmistokehittäjää kiinnostaa erityisesti virheen sattuessa, missä metodin kohdassa ongelma tarkalleen on, jotta hän pää-

see virheen jäljille ja pystyy tekemään tarvittavat korjaukset. Virheraportista ohjelmistokehittäjä pääsee suoraan hiiren klikkauksella siirtymään virheen aiheuttaneelle ohjelmakoodiriville.



Kuva 4. *Ylimpänä on näkymä IntelliJ-työkalussa suoritetuista yksikkötesteistä, joista oikean- ja vasemmanpuoliset näkymät on suurennettu lukemista helpottamaan.*

Kun ohjelmisto kehittyy laajaksi, koko yksikkötestipaketin suorittamiseen kuluu aikaa. Kuvan 4 yksikkötestien suoritusmäärää on vähennetty todelliseen tilanteeseen verrattuna, jolloin esimerkin Suite1-testipaketissa ajettavien yksikkötestien kokonaissuoritusaika on hieman alle 24 sekuntia. Tarkastellaan vertailun vuoksi koko PLP- ohjelmistokokonaisuutta, joka koostuu useasta sovelluspaketista. Koko PLP-ohjelmiston yksikkötestien

suorittaminen vie noin kymmenen minuuttia. Päättötyöhön liittyvät COM-korvauskäsittelyyn siirtyvät ulosmaksut käsitellään LifeBackOfficeBusinessTasks-sovelluksessa ja kyseisen sovelluksen yksikkötestien suoritukseen kuluu aikaa useita minuutteja.

Käytettäessä TDD-kehitystapaa, yksikkötestejä ajetaan jatkuvasti ohjelmistokehityksen iteraatiosyklin aikana, joten yksikkötestien suoritusvaiheen pitää sujua vikkelästi. Jos jokaisen pienen muutoksen jälkeen testituloksien odotteluun menee minuutteja, ohjelmistokehittäjä turhautuu ja hylkää TDD-kehitystavan käytön nopeasti. Työtä voikin nopeuttaa ajamalla vain kehityksen alla olevaan ohjelmaluokkaan liittyviä yksikkötestejä iteraatiosyklin aikana. Sovellukseen liittyvän testipaketin voi siinä tapauksessa suorittaa aikaajoin työn edetessä esimerkiksi puolen tunnin välein. Jos testipaketin suorittamisessa tulee virhe, virheen jäljitys ei muodostu mahdottomaksi, koska koodimuutoksia ei ole puolessa tunnissa ehtinyt kertyä valtavia määriä. Koko PLP-ohjelmistoon liittyvät testit voi ajaa muutaman kerran työpäivän aikana, kuitenkin viimeistään toiminnallisuuden valmistuttua.

TestNG-yksikkötestikehys mahdollistaa huomattavasti enemmän yksikkötestien ajoon liittyviä toiminnallisuuksia kuin edellä olevissa yksinkertaistetuissa esimerkeissä (ks. Kuva 3 ja Kuva 4) esitettiin. Yksikkötestejä voi esimerkiksi jakaa metoditasolla ryhmiin, jolloin samalla ryhmänimellä nimetyt, eri testiluokissa olevat metodit voidaan ajaa ryhmän nimen avulla (Beust, 2018). Ryhmän nimi määritellään testimetodin annotaation yhteydessä esimerkiksi

```
@Test(groups = {"export", "policy"},
```

jolloin määritelty metodi ajettaisiin kahden eri ryhmän yksikkötestien suorituksessa. Ryhmien nimiin viitataan tässä tapauksessa testng.xml -määrittelytiedostossa. Määrittelytiedostossa voidaan myös puuttua testien suoritusjärjestykseen tai määritellä, kuinka tarkalla tasolla yksittäisten testien suoritustulos raportoidaan. Monipuoliset yksikkötestikehyksen toiminnallisuudet mahdollistavat testien ajo-ominaisuuksien muokkaamisen kulloiseenkin tarpeeseen sopivaksi.

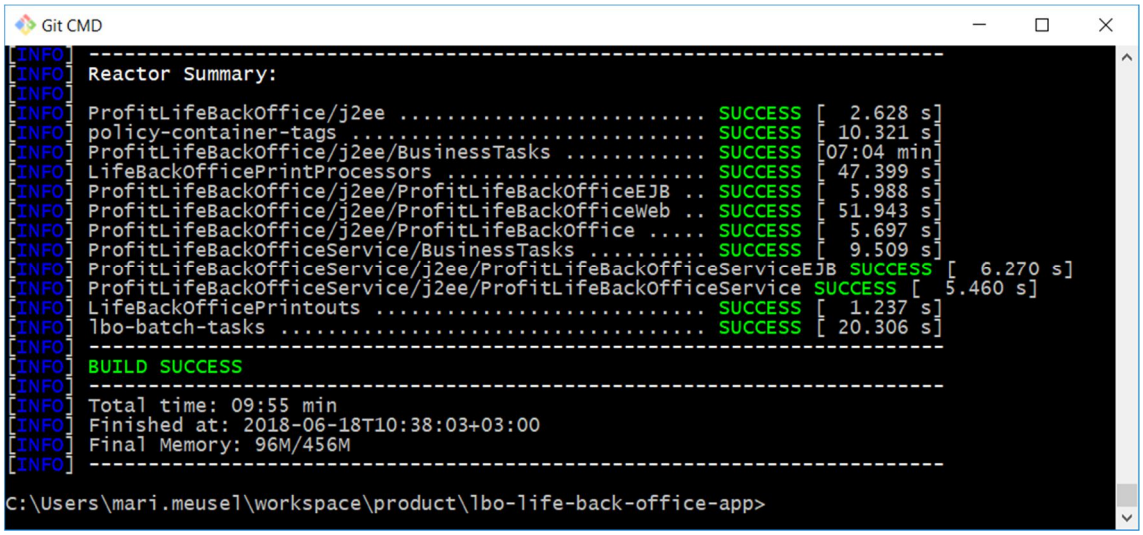
4.1.2 Testien suorittaminen ohjelmiston kääntämisen yhteydessä

PLP-ohjelmisto ja sen sisältämät sovellukset käännetään suoritettavaksi ohjelmaksi ohjelmistokehittäjän paikalliseen hakemistoon komennolla

```
mvn clean install.
```

Maven on komentorivipohjainen työkalu, joka tallentaa suoritettavan ohjelman oletusarvoisesti ohjelmistoprojektin juureen target-nimiseen kansioon ear-tiedostomuotona, joka on PLP-sovelluksen ajoympäristössä käytetyn Wildfly-sovelluspalvelimen ymmärtämä

suoritettavan sovelluksen tiedostomuoto. Maven Surefire Plugin suorittaa ohjelmistopakettiin sisältyvät yksikkötestit ohjelman kääntämisen yhteydessä ja näyttää komentoikkunassa yhteenvedon testien onnistumisesta (ks. Kuva 5).



Kuva 5. *PLP-ohjelmiston kääntäminen suoritettavaksi ohjelmistoksi ja testien suoritus Maven ja Maven Surefire Plugin -työkalujen avulla*

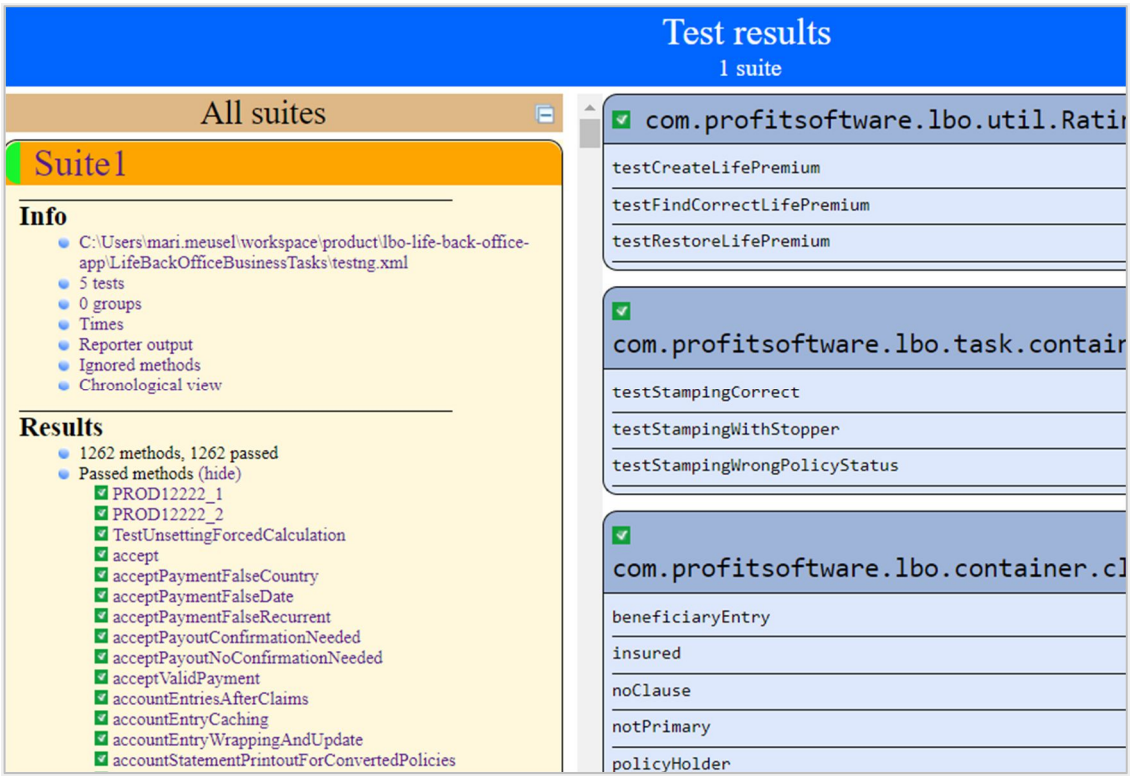
Kun ohjelmistokehittäjä työstää uusia ominaisuuksia, kehitystyön aikana on tarpeen testata toimintoa loppukäyttäjän käyttöliittymästä eli web-selaimella. Testausta varten ohjelmistokehittäjän tarvitsee kääntää ohjelmisto ja siirtää se virtuaalikoneelle suoritettavaksi. Riippuen toteutettavasta toiminnosta, tätä käännös- ja siirtorutiinia voi joutua toistamaan usein. Testien ohittaminen ohjelmistopaketoinnin yhteydessä onnistuu komennolla

```
mvn -DskipTests install,
```

joka on hyödyllinen käsky ajan säästämiseksi. Wildfly-sovelluspalvelin tarvitsee kääntämisen lisäksi muutaman minuutin saattaakseen uuden ohjelmistopaketin suorituskuntoon. Kaikki yksikkötestit kannattaa kuitenkin suorittaa aika ajoin ja viimeistään silloin, kun toiminnallisuus on valmis.

Maven Surefire Plugin muodostaa testiraportit oletusarvoisesti txt (teksti) ja xml -muotoisina jokaisen ohjelmistopakettiin sisältyvän sovelluksen omaan target-kansioon. Jos käytössä on Maven Surefire Report Plugin, muodostuu ohjelmiston kääntämisen yhteydessä html-muotoinen tiedosto, jota on helppo lukea web-selaimen avulla (ks. Kuva 6).

Web-selaimessa näytettävä raportti on helposti tulkittavissa silmämääräisesti. Kuvan 6 vasemmalla puolella näkyy testien yhteenveto ja suoritetut metodit. Tässä tapauksessa tarkastellaan korvausmaksukäsittelyt sisältävän LifeBackOfficeBusinessTasks-sovelluksen testiraporttia, jossa on suoritettu sovellukseen sisältyvät kaikki yksikkötestit Kuvassa 3 esitellyn suppeamman testikoosteen sijaan.



Kuva 6. *Osa Maven Surefire Report Plugin -työkalun avulla muodostetusta testiraportista*

Raportista nähdään, että testiryhmiä on kaikkiaan 5 ja niissä on yhteensä 1262 testimetodia. Testien suoritus on onnistunut kaikkien yksikkötestien osalta.

4.2 TDD-kehitysmenetelmän toteuttaminen käytännön ohjelmointityössä

Yksikkötestauksen ja TDD-kehitysmenetelmän oppiminen vei päättötyön toteuttajalta aikaa toteutustyössä, koska työn toteuttajalla ei ollut entuudestaan kokemusta kyseisistä tekniikoista. Työssä keskityttiin pilotoimaan TDD-työskentelytapaa niissä osissa ohjelmistokehitystä, joissa tuotettiin kokonaan uutta ohjelmakoodia, jotta työ kokonaisuudessaan pysyi muun toteutustiimin aikataulussa. Näin ollen TDD-menetelmää käytettiin osissa, joissa hyödynnetään COM-ohjelmiston siirtorajapinnan toteuttavia, JAXB-mallin sitomisen kautta syntyneitä ohjelmaluokkia. Maksumääräysoliot toteutetaan näihin ohjelmaluokkiin perustuen ja kyseisten olioiden tilaa testataan yksikkötesteissä. Tässä luvussa esitettyjä ohjelmaesimerkkejä on hieman muuteltu todellisesta toteutuksesta yrityssalaisuuden turvaamiseksi.

4.2.1 Yksikkötestien toteutus

Yksi toteutettavan ohjelmiston päätoiminnallisuuksista on korvausmaksujen poiminta tietokannasta. Tietokantahaku palauttaa ohjelmalle poimitut tietueet, joista muodostetaan

rajapinnan toteuttavat maksumääräysviennit. Yksikkötestien perusajatukseen kuitenkin kuuluu se, että testit eivät käytä apunaan tietokantaa eivätkä luo tietokantayhteyttä (ks. 2.1 Yksikkötestaus). Tietokannassa olevat, poimintaa odottavat maksumääräykset vaihtelevat ajankohdasta ja muista ohjelmiston tiloista riippuen. Tästä johtuen maksumääräysten poiminta tietokannasta ei olisikaan järkevää yksikkötestien kannalta, koska yksikkötestissä testattavan olion tila halutaan olevan aina sama riippumatta testien suorittamisajankohdasta.

Koska tietokantapoimintaa ei suoriteta, Ohjelman 5 yksikkötestiesimerkissä muodostetaan kaksi kuvitteellista, testiä varten perustettavaa maksuoliota (payment-olio) ja niiden tilat alustetaan halutuilla arvoilla. Ohjelmassa 5 kolmella pisteellä ilmaistaan piilossa olevaa kooditoteutusta.

```
    ...
44  @BeforeClass
    public void setUp() throws Exception {
46    ...
    }
59
    @Test
61  public void testPaymentOrderExportCom() throws Exception {
      ...
95    Assert.assertEquals(filedata.getPaymentMessage().get(0)
            .getPaymentOrder().getPolicyTaxCode(),
97          PolicyTaxCode.fromValue("S"));
      ...
106 }

108 private List<Payment> createPayments() {
    return Arrays.asList(
110         createPayment(today, "S", 1200.0),
            createPayment(yesterday, "U", 183.8))
112 }

114 private Payment createPayment(double grossAmount, String taxCode,
      Date paymentDate) {
116
      Payment payment = new Payment();
118   payment.setPaymentDate(paymentDate);
      payment.setCode(taxCode);
120   payment.setAmount(grossAmount);
      ...
144   return payment;
    }
```

Ohjelma 5. *Yksikkötestiesimerkki maksumääräysolion testaamiseksi*

Testin aluksi Ohjelman 5 *@BeforeClass*-annotaatiolla merkityssä setUp()-metodissa, ennen varsinaisten testien suorittamista, alustetaan yksikkötestausympäristö halutuilla arvoilla. Metodin ohjelmakoodia ei näytetä esimerkissä, mutta alustuksessa suoritetaan createPayments()-metodi, joka luo kaksi maksuoliota ja palauttaa ne listamuotoisena kut-

sujalle. Maksuoliot välitetään luokalle, joka luo COM-ohjelmiston siirtorajapinnan määritysten mukaisen maksumääräysolion ja palauttaa sen filedata-nimisenä oliona testattavaksi.

Testialustuksessa, eli setUp()-metodissa syntynyttä filedata-maksumääräysoliota voi testata *@Test*-annotaatiolla merkityssä metodissa ilman, että olioita tarvitsee kirjoittaa ja tallentaa lopullisiksi xml-muotoisiksi tiedostoiksi. Olio sisältää koko maksumääräykselle yhteisiä tietoja sekä maksumääräyksen sisällä olevalle jokaiselle maksulle vain kyseistä maksua koskevia tietoja (ks. 3.4.1 XML-rajapinta). Ohjelman 5 yksikkötestissä verrataan maksumääräyksessä ensimmäisen maksun, johon viitataan indeksillä 0 (get(0)), verokoodia tekstiarvoon ’S’. Koska createPayments()-metodissa ensimmäinen maksu alustetaan verokoodin arvolla ’S’, assertEquals()-metodin testituloksen oletetaan palauttavan tosituloksen eli verrattavien arvojen oletetaan olevan samanlaiset.

Erittäin tärkeä toiminnallisuus rajapinnan toteutuksessa on XML-tiedoston syntyminen ja tallentuminen tiedostojärjestelmään. Koska yksikkötestien ei tule käyttää tiedostojärjestelmää apunaan (ks. 2.1 Yksikkötestaus), yksikkötestillä ei voida todentaa, muodostuiko maksumääräystiedosto vai ei. Maksumääräysolion tallentaminen JAXB-ohjelmointirajapinnan marshalling-tekniikalla toteutetaan ohjelmistossa seuraavalla tavalla:

```
JAXBContext jaxbContext = JAXBContext
.newInstance(BeneficiaryMessagesFileType.class);
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(fileData, filename);
```

Kyseinen ohjelmaosuus tulee hoitaa toteutuksessa siten, että mahdolliset ongelmat XML-tiedoston muodostuksessa ja tallennuksessa huomataan ohjelman ollessa tuotantokäytössä. Ohjelmaosuus tuleekin sijoittaa try - catch -virheenkäsittelyyn, joka on karkeasti kuvattu esimerkissä:

```
try {
      /* Yritä kirjoittaa XML-tiedosto levylle */
} catch (JAXBException e) {
      /** Tiedoston kirjoitus epäonnistui. Kirjoita virheilmoitus
      * tapahtumalokiin
      */
}
```

Catch-virheenkäsittelyosiossa hoidetaan tiedostokirjoituksen epäonnistuttua virheen raportointi ohjelmiston käyttäjien tiedoksi.

4.2.2 Työskentely TDD-menetelmällä

Oletetaan ohjelmakoodin olevan siinä vaiheessa, että Ohjelman 5 yksikkötestit menevät läpi. Testattava ohjelmatoteutus on tässä vaiheessa oltava vähintään sellainen, että se toteuttaa maksujen viennin rajapintamääritysten mukaisesti ainakin verokoodin osalta.

Koska maksuista halutaan viedä COM-ohjelmistoon vielä paljon muitakin tietoja, ohjelmistokehittäjä jatkaa työskentelyään TDD-ohjelmistokehityssyklin mukaisesti (ks. 2.5.1 Testivetoinen ohjelmistokehityssykli). Seuraavassa viidessä kohdassa käydään läpi syklin vaiheet erään toiminnallisuuden toteutusprosessissa.

1. Kirjoita yksikkötesti uudelle, toteuttamattomalle toiminnolle.

Ohjelmistokehittäjä valitsee seuraavan toteutettavan toiminnallisuuden ohjelmistomäärittelydokumentista. Määrittelyssä todetaan, että maksuun halutaan liittää tieto, onko kyseessä korvauksen ensimmäinen maksuerä. Jos erä on ensimmäinen, eli yhtään aiempaa maksua kyseiseen korvaukseen liittyen asiakkaalle ei ole maksettu, maksun firstPayment-attribuutin arvoksi annetaan tosi, muussa tapauksessa epätosi. Tämä toiminnallisuus voidaan testata lisäämällä seuraavat kaksi riviä Ohjelman 5 riville 98, testimetodiin testPaymentOrderExportCom():

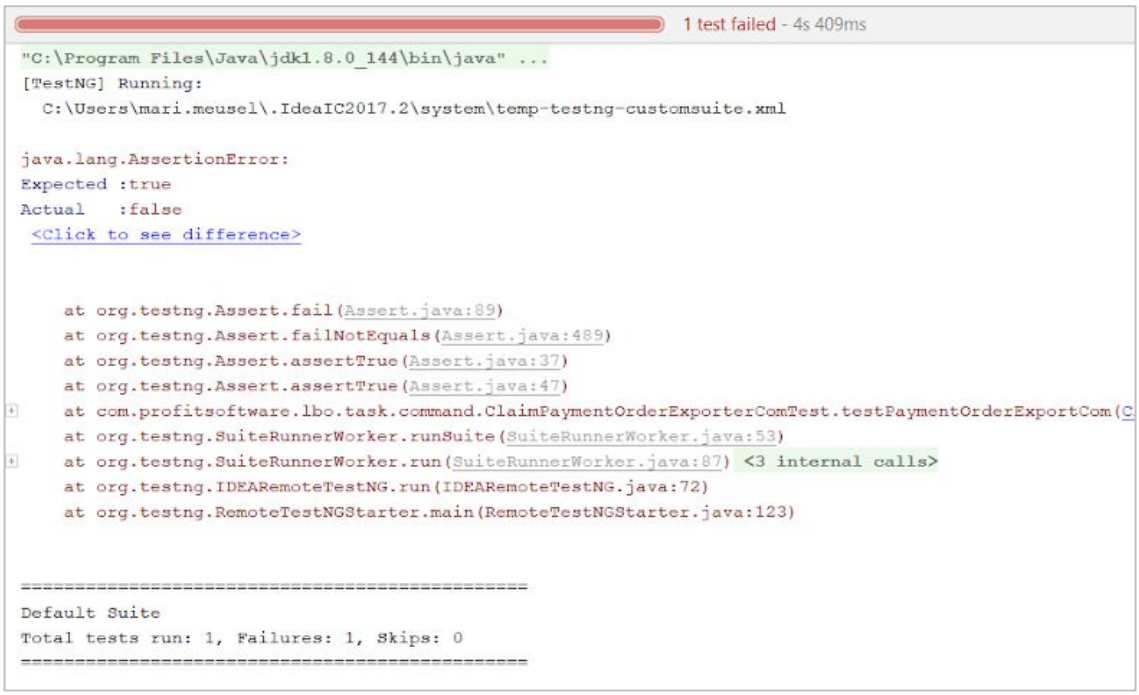
```
Assert.assertFalse(filedata.getPaymentMessage().get(0).getPaymentOr-
der().isFirstPayment());
Assert.assertTrue(filedata.getPaymentMessage().get(1).getPaymentOr-
der().isFirstPayment());
```

Kyseisessä yksikkötestissä oletetaan maksumääräysolion sisällä olevan ensimmäisen maksun (indeksi 0) olevan korvausmaksun jatkoerä, eli firstPayment-attribuutin arvo on epätosi. Maksumääräysolion sisällä olevan, järjestyksessä toisen (indeksi 1) maksun oletetaan olevan korvausmaksatuksen ensimmäinen maksuerä eli firstPayment-attribuutin arvona on true.

2. Suorita aiemmin lisätyt testit sekä kohdassa 1 lisätyt uudet testit ja varmista, että uudet testit eivät mene läpi.

Suoritetaan Ohjelman 5 yksikkötestikokonaisuus IntelliJ-ympäristössä. Kuten on oletettavissa, yksikkötestit eivät onnistu uusien yksikkötestien osalta, koska toiminnallisuuden toteuttavaa ohjelmaosuutta ei vielä ole. Kun tosi ja epätosi -arvoja testataan, kannattaa aina testata molemmat arvot. Riippuen ohjelmatoteutuksesta, tässä tapauksessa COM-rajapinnan toteuttavien ohjelmaluokkien toteutuksesta, totuusarvoksi voi alustua oletuksena vaihtoehtoisesti arvo tosi, epätosi tai null ilman, että olion attribuutiksi asetetaan mitään luokkaa käyttävässä ohjelmakoodissa.

Suoritettujen yksikkötestien tulos näkyy Kuvassa 7, jonka yhteydessä huomataan, että yksikkötesti epäonnistuu vasta isFirstPayment()-metodin tosi-arvoa testaavassa yksikkötestissä. Tämän perusteella näyttää siltä, että totuusarvo tässä ohjelmatoteutuksessa on oletuksena epätosi.



Kuva 7. *Yksikkötestin suorituksen tulos kun yksikkötesti on toteutettu, mutta ohjelmakooditoteutus puuttuu.*

Nyt kun uudet yksikkötestit on suoritettu ja niistä on saatu epäonnistunut testitulos, on varmistettu, että ensimmäisen maksuerän ilmaisevaa totuusarvoa ei vielä ole ohjelmassa toteutettu. Voidaan turvallisesti siirtyä TDD-kehityssyklin seuraavaan vaiheeseen.

3. Kirjoita uutta toimintoa vastaava ohjelmakoodi.

Toteutusvaiheessa huomataan, että tietokantakyselyä pitää täydentää, jotta tieto ensimmäisestä maksuerästä saadaan mukaan poimintaan. Tietokantakyselyyn lisätään mukaan tieto, koska korvausmaksun ensimmäinen erä on tai on ollut ja kyseinen päivämäärä tallennetaan maksuolion firstPaymentDate-attribuutin arvoksi. Tätä päivämäärää verrataan kyseessä olevan maksuerän periodStartDate-attribuutin päivämääräarvoon. Mikäli firstPaymentDate-päivämäärä on ennen periodStartDate-päivämäärää, kyseessä ei ole ensimmäinen maksuerä. Sama päivämäärä vertailussa tarkoittaa, että kyseessä on ensimmäinen maksuerä ja firstPayment-attribuutti on tällöin tosi. Maksumääräysolion sisällä olevassa maksuoliossa firstPayment-attribuutin arvo asetetaan seuraavalla tavalla ehtolauseessa:

```
if (payment.getFirstPaymentDate()
      .before(payment.getPeriodStartDate){
     paymentOrderType.setFirstPayment(false);
} else {
      paymentOrderType.setFirstPayment(true);
}
```

4. Aja kaikki yksikkötestit uudelleen ja tarkista, että myös uudet testit menevät läpi.

Kooditoteutuksen jälkeen huomataan, että yksikkötestissä ei anneta ollenkaan vertailussa käytettäviä päivämääriä, jonka vuoksi yksikkötesti epäonnistuu

```
java.lang.NullPointerException
```

poikkeusilmoituksella. Null-arvoa ei voi vertailla before()-metodilla. Yksikkötestiä pitää vielä valmistella createPayments() ja createPayment() -metodien osalta, jotta vertailtavat päivämäärät saadaan testiin mukaan. Metodi createPayment() muutetaan vastaanottamaan päivämäärät parametrina sekä asettamaan päivämäärät metodissa maksuolion periodStartDate ja firstPaymentDate attribuuttien arvoiksi. Kun createPayment()-metodia kutsutaan createPayments()-metodissa, rivin 110 parametreihin lisätään arvot, joissa firstPaymentDate-päivämäärä on ennen periodStartDate-päivämäärää, jolloin isFirstPayment()-metodin oletetaan palauttavan arvon epätosi. Rivin 111 parametreihin lisätään kaksi samaa päivämäärää, jolloin isFirstPayment()-metodin oletetaan palauttavan arvon tosi.

Jotta TDD-syklin kohdassa 2 tehty varmistus siitä, että testit eivät mene läpi voidaan todentaa, kommentoidaan kohdassa 3 tehty kooditoteutus pois suoritettavasta ohjelmakoodista ja ajetaan yksikkötestit. Nyt testien suoritus pitäisi edelleen epäonnistua. Kun tämä on varmistettu, koodikommentit voidaan poistaa ja siirtyä uudelleen kohtaan 4 suorittamaan yksikkötestit. Kun todetaan, että yksikkötestit menevät läpi, voidaan siirtyä kohtaan 5.

5. Refaktoroi ohjelmakoodia.

Nyt on aika parantaa ohjelmakoodin sisäistä rakennetta. Koska syklin kohdassa 4 huomattiin, että päivämäärien Null-arvot aiheuttavat ongelmia, kooditoteutusta täydennetään tällaisten ongelmien varalta. Ohjelmakoodi pysyy myös luettavampana ja selkeämpänä, kun tällaiset vertailua vaativat ohjelmaosiot siirretään omaksi metodikseen. Näin syntyy uusi metodi isFirstPayment(), jonka ohjelmatoteutus on:

```
public boolean isFirstPayment(Payment payment) {
  if (payment.getFirstPaymentDate() == null
      || payment.getFirstPaymentDate() == null) {
     return true;
  }
  return !payment.getFirstPaymentDate()
      .before(payment.getPeriodStartDate());
}
```

Nyt maksumääräysolion firstPayment-attribuutin asettaminen on virtaviivaisempaa:

```
paymentOrderType.setFirstPayment(isFirstPayment(payment));
```

Muutosten jälkeen ajetaan jälleen yksikkötestit ja varmistetaan, että testien suoritus onnistuu edelleen. Jos näin on, voidaan alkaa TDD-kehityssykli alusta uuden toiminnallisuuden kanssa.

4.3 Johtopäätökset

Testivetoinen ohjelmistokehitys kehitysmenetelmänä soveltuu hyvin sellaisiin tapauksiin, jossa kehitetään täysin uutta ohjelmakoodia. Mikäli kyseessä on jo olemassa olevan toteutuksen laajennus, TDD-menetelmää voidaan käyttää silloin, kun olemassa olevaan toteutukseen liittyvien yksikkötestien kattavuus on korkea. Testivetoisen ohjelmistokehityssyklin olettamus on, että tehtyjen muutosten jälkeen aiemmat yksikkötestit paljastavat, mikäli uusi toiminnallisuus aiheutti ongelmia jossain jo aiemmin toteutetussa toiminnallisuudessa. Mikäli yksikkötestien toteutuksessa on puutteita, aiemman ohjelmistototeutuksen refaktoroinnissa pitää olla varovaisempi. Tämän päättötyön yhteydessä toteutettu toiminnallisuus sisältää runsaasti riippuvuuksia aiempaan koodipohjaan, jonka vuoksi TDD-kehitysmenetelmän pilotointiin valittiin ohjelmaluokka, joka luotiin kokonaisuudessaan työn yhteydessä.

Tarkasteltaessa TDD-kehityssyklin käyttöä todellisen ohjelmointiesimerkin kautta (ks. 4.2.2 Työskentely TDD-menetelmällä) huomattiin, että ohjelmointityössä ei aina voida noudattaa kehityssyklin järjestystä puhdasoppisesti. Esimerkissä jouduttiin palaamaan kooditoteutuksen jälkeen takaisin yksikkötestin muokkaukseen. Kehityssyklin tarkoitus onkin ohjata kehitystyötä eikä sitä tarvitse noudattaa jatkuvasti orjallisesti. Työn edetessä ohjelmakoodin refaktorointia toteutettiin työn eri vaiheissa myös niihin toiminnallisuuden osiin, jotka eivät olleet kyseisessä tilanteessa työn alla. Näissä tapauksissa olemassa olevat yksikkötestit paljastivat hyvin, mikäli refaktorointi aiheutti toiminnallisia muutoksia.

Noudatettaessa TDD-kehitysmenetelmää, menetelmän etuja (ks. 2.5.2 TDD-kehitysmenetelmän edut) ei ole helppo arvioida verrattuna TLD-kehitysmenetelmään. Tämän päättötyön ohjelmistoprojektin toteutustyön kokemusten perusteella TDD-menetelmän eduksi voidaan todeta, että se ohjaa toteutettavan ohjelmakoodin rakennetta yksikkötestikelpoiseksi. Koska yksikkötesti kirjoitetaan etukäteen, varsinainen ohjelmakoodi on kapseloitava metodeihin siten, että kukin metodi toteuttaa vain yhtä tehtävää. Metodien mahdollisimman yksinkertainen toteutus ja metodin toiminnallisuuden rajoittaminen yhteen tehtävään on yksi Enzlerin esittämä puhtaan ohjelmakoodin päämääriä ja sen ansiosta ohjelmakoodin toiminnallisuuden myöhempi tulkinta on helpompaa (Enzler, 2014).

Päättötyön toteuttaja koki erityisesti menetelmän käytön etuna, että ohjelmistotyöskentely nopeutui, kun kehitteillä olevia toimintoja pystyi kokeilemaan suoraan IntelliJ-kehitysympäristössä ajamalla testejä syntyvää ohjelmakoodia vasten (ks. 2.5.2 TDD-kehitysmenetelmän edut, kohta 2). Yksikkötestien suorittamiseen kului aikaa muutama sekunti ja kehittäjä näki, toimiko kehityksen alla oleva ohjelmakoodi halutusti vai vaatiko se vielä

muutoksia. Mikäli yksikkötestejä ei olisi toteutettu testivetoisesti, jokaisen uuden toiminnallisuuden testaaminen olisi pitänyt tehdä PLP-sovelluksen ajoympäristössä virtuaalikoneella. Sitä varten sovellus olisi pitänyt kääntää ja siirtää virtuaalikoneelle suoritettavaksi. Sen jälkeen olisi pitänyt suorittaa tarvittavat tehtävät käyttöliittymän kautta, jotta järjestelmästä löytyy korvaustilassa oleva COM-järjestelmään siirtyvä maksumääräys sekä suorittaa maksujen poiminta-ajo XML-siirtotiedoston syntymiseksi. Kun ohjelmistokehittäjä pääsee lopulta tarkistamaan siirtotiedostosta, siirtyivätkö sinne oikeat tiedot, aikaa on kulunut useita minuutteja.

Yksikkötestien kirjoittamiseen tarvittava aika vaihtelee. Jotta Ohjelman 5 testiluokka saatiin perustettua ja asetettua siihen kuntoon kuin se esimerkissä on, tarvittiin paljon työtä ohjelman suoritusympäristön alustamiseksi setUp()-metodissa. Koska yksikkötestit on tarkoitus suorittaa niin sanotusti ’puhtaalta pöydältä’, tarvitsee ajoympäristö alustaa muun muassa käytettävän kielen, parametriston ja testin suorituspäivämäärän emuloinnilla. Kun Ohjelman 5 testimetodiin kirjoitetaan uusia yksikkötestejä, kuten 4.2.2 *Työskentely TDD-menetelmällä* -kappaleessa tehtiin, yksikkötestin lisääminen sujui yleensä nopeasti.

Koska yksikkötestiympäristön alustaminen on työlästä, ohjelmistokehittäjän omat yksilölliset taipumukset tai aikataulukiireet ohjaavat yksikkötestien toteuttamisvolyymia. Perustuen päättötyön toteuttajan omiin ja keskusteluihin tilaavan yrityksen ohjelmistokehittäjien kanssa, koetaan monesti toteutettavan työn olevan valmis, kun toiminnallisuuden toteuttava ohjelmakoodiosuus on valmis. Yksikkötestien toteutus jälkikäteen koetaan lisätyöksi, josta halutaan mahdollisuuden mukaan tinkiä. TDD-menetelmää käytettäessä yksikkötestit toteutetaan rinnakkain ohjelmistotoiminnallisuuksien toteutuksen kanssa, jolloin testien toteuttaminen ei jää erilliseksi tehtäväksi.

Mikäli uudet toiminnallisuudet hyväksytään koodikatselmoinnin yhteydessä ilman yksikkötestejä, yksikkötestien puuttuminen ohjaa yksikkötestien syntymistä myös toiminnon jatkokehityksessä. Perustuen päättötyön tilanneen yrityksen ohjelmistokehittäjien haastatteluihin, he eivät yleensä toteuta tällaiseen ohjelmistoluokkaan yksikkötestejä jatkossakaan, koska he joutuisivat perustamaan koko yksikkötestiympäristön kyseiselle toiminnallisuudelle. Pienen toiminnallisuuden muutos ja siihen liittyvä yksikkötestin asettaminen aiheuttaisi huomattavan työmäärän verrattuna pelkkään toiminnallisuuden muutoksen työmäärään. Myös yksikkötestien toteuttaminen jälkikäteen voi olla kompleksista tai jopa mahdotonta. Jos testauksen alla olevan ohjelmiston toteutuksessa ei olla otettu yksikkötestausta huomioon, ohjelmiston rakenne voi estää yksikkötestien toteuttamisen.

Ohjelmistoyrityksen oma linjaus yksikkötestien tärkeydestä sekä resurssointi ohjaa ohjelmistokehittäjien työtä. Koodikatselmoinnissa uusia toimintoja ei tule hyväksyä, ellei yksikkötestit ole toteutettu yrityksen haluamalla koodikattavuudella. Sen vuoksi koodikatselmoinnissa tulisikin käyttää henkilöitä, joilla on yrityksen linjaus tiedossa ja tarvittava kokemus yksikkötestauksesta.

Luvussa 2 *Testausmenetelmät ohjelmistokehitys* esiteltiin eri testausmenetelmiä ja todettiin niillä olevan oma roolinsa ohjelmiston kokonaislaadun varmistamisessa. Yksikkötestauksella, toteutetaan se sitten testivetoisesti tai ohjelmatoiminnallisuuksien toteuttamisen jälkeen, on eri testimenetelmien kirjossa oma paikkansa ja sen tehtävä on testata yksittäisiä Java-luokkien sisällä olevia toiminnallisuuksia. Kuten tässäkin ohjelmistoprojektissa, PLP-ohjelmiston liiketoimintalogiikka toteutetaan monessa tapauksessa tietokantakyselyillä, jotka tapahtuvat tietokannan sisällä aliohjelmina (SQL procedures). Tietokantakyselyjen toimimista oikein ei voida testata yksikkötestausmenetelmillä, vaan avuksi tarvitaan integraatiotestausmenetelmiä. Tässä päättötyössä toteutettu ohjelmaosuus voidaan testata automaatio- tai manuaalitestauksessa luomalla COM-korvausmaksatuksen sisältävä sopimus ja saattamalla se korvaustilaan. Testauksen yhteydessä suoritetaan tarvittavat ajastetut tehtävät korvausmaksun sisältävän XML-muotoisen siirtotiedoston luomiseksi. Tämän jälkeen tarkistetaan tiedostojärjestelmästä, muodostuiko sinne siirtotiedosto ja sisältääkö se testattavan korvausmaksumääräyksen.

5. YHTEENVETO

Tämän päättötyön tarkastelunäkökulmaksi ja ohjelmistokehitystavaksi valitulla testivetoisella ohjelmistokehitystavalla haluttiin tutkia kehitystavan soveltuvuutta päättötyön tilanneen yrityksen ohjelmistokehitysympäristöön ja työskentelytapoihin. TDD-menetelmällä tuotetaan yksikkötestejä, joilla varmistetaan ohjelmiston sisäistä, eli ohjelmakoodin laatua, selkeyttä ja jatkokehitettävyyttä. Työssä toteutettiin rajapinta tilaavan yrityksen kahden oman ohjelmistotuotteen välille, käytettiin TDD-kehitysmenetelmää ohjelmistokehityksen valittuihin osiin sekä arvioitiin kehitysmenetelmän hyötyjä. Työskentelyssä arvioitiin myös yksikkötestien nykyistä tilaa ja kehitysprosesseja, puuttumatta siihen, millä kehitysmenetelmällä ne ovat syntyneet.

Taulukko 1. *Tiivistelmä päättötyön lähtökohdista, työskentelyprosessista ja keskeisistä tuloksista*

Tavoite	Arvioida TDD-kehitysmenetelmän soveltuvuutta tilaavan yrityksen ohjelmistokehitystyöhön.
Aika	Tammikuu 2018 - elokuu 2018
Työryhmä	Työhön liittyvän ohjelmakooditoteutuksen ja siihen liittyvät yksikkötestit toteutti päättötyön tekijä. Tilaajan asiantuntijat toimivat teknisen työn tukena sekä antoivat tietoja nykyisten työprosessien kulusta.
Työn arviointi	Työhön osallistunut ohjelmistokehitystiimi ratkoi työn toteuttajan kanssa työn aikana esiin tulleita ongelmia ja katselmoi syntyneet koodit. Prosessikehitystiimi osallistui yksikkötestaukseen liittyvien oivallusten arviointiin. Päättötyön ohjaaja tilaajayrityksessä sekä päättötyön tarkastaja yliopistossa osallistuivat kirjallisen työn rakenteen ja oikeellisuuden arviointiin.
Tulokset	Työhön liittyvä ohjelmiston toiminnallisuus toteutettiin ja valituissa osissa käytettiin TDD-kehitysmenetelmää. TDD-kehitysmenetelmän etuja verrattuna TLD-menetelmään ei kaikilta osin voitu todistaa oikeaksi. Kehitystyön iterointisykliä TDD-kehitystapa nopeutti huomattavasti sekä ohjasi syntyvää koodia yksikkötestikelpoiseksi.

Suurimmat muutokset	Yksikkötestien kattavuutta uusien ja jo olemassa olevien tärkeimpien toiminnallisuuksien yksikkötestien kattavuutta päätettiin lisätä. Yritys selkeyttää vaatimuksiaan yksikkötestien kattavuuden suhteen. Koodikatselmoinneissa käytetään henkilöitä, jotka katselmoivat myös syntyneet yksikkötestit sekä arvioivat niiden rakennetta ja kattavuutta. Ohjelmistokehittäjille lisätään koulutusta, jotta he saavat tarvittavat valmiudet yksikkötestityöhön.
Suurimmat ongelmat	Nykyisen ohjelmistokoodin puutteet paikoitellen yksikkötestien kattavuudessa.

Yksikkötestit ovat vain yksi testimenetelmä muiden menetelmien joukossa. Yksikkötesti mittaa yhden pienen ohjelmiston osan toimivuutta, jonka avuksi tarvitaan muita testimenetelmiä mittaamaan isompien kokonaisuuksien toimivuutta. PLP-ohjelmistossa liiketoimintalogiikka on rakennettu monesti tietokannan sisälle SQL-proseduureihin ja myös niiden toiminta pitää varmentaa muilla kuin yksikkötestimenetelmillä.

LÄHTEET

Acharya, S. (2013). Test-Driven Development with Mockito, Packt Publishing, Olton.

Beust, C. Documentation, TestNG. Saatavissa (accessed 24/05): http://testng.org/doc/documentation-main.html.

Enzler, U. (2014). Clean ATDD/TDD Cheat Sheet, bbv Software Services.

Fallside, D.C. & Walmsley, P. XML Schema Part 0: Primer Second Edition, W3C. Saatavissa (accessed 01/06): https://www.w3.org/TR/xmlschema-0/.

Horch, J.W. (2004). Test-Driven Development-By Example, Software Quality Professional, Vol. 6(3), pp. 43.

Leung, H.K.N. & White, L. (1989). Insights into regression testing (software testing), IEEE Comput. Soc. Press, pp. 60-69.

Mackinnon, T., Freeman, S. & Craig, P. (2000). Endo-Testing: Unit Testing with Mock Objects, CiteSeerX.

Martin, R.C. The Cycles of TDD. Saatavissa (accessed 18.3.2018): http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html.

Martin, R.C. (2007). Professionalism and Test-Driven Development, IEEE Software.

Mehta, B. & Ort, E. Java Architecture for XML Binding (JAXB), Oracle Technology Network. Saatavissa (accessed 01/06/2018): http://www.oracle.com/technetwork/articles/javase/index-140168.html.

Patni, S. (2017). Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS, Apress.

Shore, J. Red-Green-Refactor. Saatavissa (accessed 4.4.2018): http://www.jamesshore.com/Blog/Red-Green-Refactor.html.

Sprengers, J. Mocks or the real thing? Tips for better unit testing. Saatavissa (accessed 06/28): https://blog.codecentric.de/en/2017/07/mocks-real-thing-tips-better-unit-testing.

Sujoy, A. (2014). Mastering Unit Testing Using Mockito and JUnit, ProQuest Ebook Central.

The Apache Software Foundation Apache Maven Project, Maven Introduction. Saatavissa (accessed 05/03): https://maven.apache.org/what-is-maven.html.

Thomas, D. & Hunt, A. (2002). Mock objects, IEEE Software, Vol. 19(3), pp. 22-24.

Vázquez, G. An Introduction to API's. Saatavissa (accessed 06/14): https://restful.io/an-introduction-to-api-s-cee90581ca1b.

Vestola, M. (2008). TDD:n edut – tarua vai totta? Teknillinen korkeakoulu, kandidaatintyö.

WebSystique TestNG dependsOnMethods Example. Saatavissa (accessed 03/12): http://websystique.com/java/testing/testng-dependsonmethods-example/.

Wikipedia List of unit testing frameworks. Saatavissa (accessed 04/26): https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java.